BAYESIAN-HARNESS: PROBABILISTIC KNOWLEDGE CONTROL FOR LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

An agent working on long-term tasks usually serialises its knowledge of, e.g., tools, interfaces, and codebases in its local environment. However, knowledge becomes stale and expires as the internal or external environment changes. Many existing harnesses update memory records after encountering an error, without explicitly modelling how knowledge expires over time. In this paper, we therefore model knowledge staleness as a generative process from a Bayesian perspective. We maintain two latent random variables for each memory, validity and quality. The validity is a binary latent state that fails at a Markov-modulated hazard and is restored only by a rewrite. We model quality in Bayesian form with a Beta-Bernoulli model over the memory's success rate conditional on validity, with memories sharing statistical strength within each subsystem pool. We further build two datasets, MASKEDMIG and ACMESDK. MASKEDMIG extracts 157 real API changes from 18 Python libraries, each removing an old API and adding a named replacement in a release between April 2025 and August 2026. ACMESDK is a company Software Development Kit (SDK) benchmark. It is based on SOP-Bench and generated by Claude Code (Fable 5.1). Through extensive empirical investigation, from a 4B model to 27B and two closed-weight Gemini models, Bayesian-Harness is the best maintenance policy on every model of MASKEDMIG, 3.1–8.3 points above the strongest baseline of each, and in all 15 model–budget configurations of ACMESDK.

Anonymous code and datasets

`github.com/anonymouspaper987/bayesian_harness`

`github.com/anonymouspaper987/bayesian_harness_datasets`

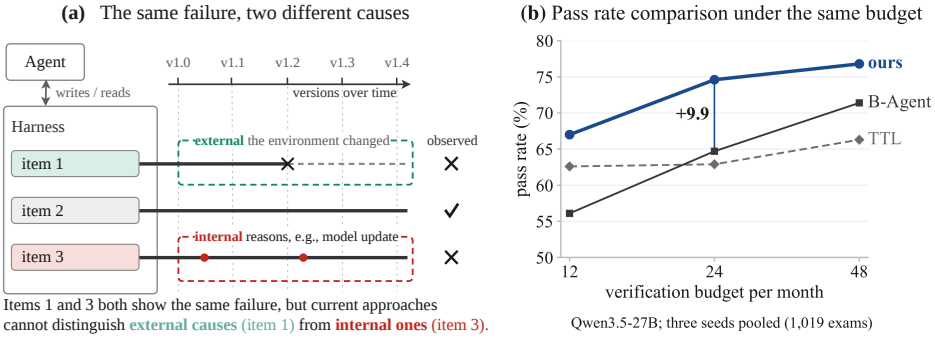


Figure 1: (a) Three memory items of a harness over five releases. Item 1 fails after the environment changes, so the cause is *external*; item 3 never goes stale but was written with low quality, so the cause is *internal*; item 2 remains good. (b) All ACMESDK monthly tests by budget, Qwen3.5-27B, three seeds pooled (Table 6).

1 INTRODUCTION

Agents based on Large Language Models (LLMs) have emerged as a powerful paradigm for building autonomous systems that can reason, plan, and interact with external environments (Yao et al., 2023; Park et al., 2023; Wang et al., 2024a;b; Gonzalez-Pumariega et al., 2026). From a high-level and

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

abstract perspective, an agent consists of one or more LLMs and a harness. LLMs provide reasoning capabilities for processing each input and produce outputs that are understandable to humans. The harness provides a platform for the LLMs to interact with tools (Guo et al., 2025), environments, and external knowledge (Mei et al., 2025; Qi et al., 2026; Ning et al., 2026; Wei et al., 2026).

Existing harness management approaches commonly keep *memory items* or *knowledge items* (i.e., discrete, addressable records of information the agent has acquired during operation) until the context fills up and then compress them, refresh the oldest memory, or discard memories based on observed outcomes (Packer et al., 2023; Chhikara et al., 2025; Xu et al., 2025; Li et al., 2025; Rasmussen et al., 2025; Zhao et al., 2024; Ouyang et al., 2026b; Fang et al., 2026; Ouyang et al., 2026a). Time-aware methods such as MemoryBank's forgetting curve (Zhong et al., 2024) and other time-driven schemes (Lam et al., 2026; Ji et al., 2026) model how memory decays over time. Bayesian-Agent (Wu et al., 2026) models memory reliability with a feature-conditioned categorical posterior updated after each use.

These methods generally assign a single score to each memory item, making it difficult to distinguish failures caused by low-quality memories from those caused by environmental changes. Recent benchmarks and studies further show that agents often fail to revise memories after they become invalid, regardless of the cause (Chao et al., 2026; Patel, 2026; Xu et al., 2026). This raises a question: ***Was this memory already bad (internal), or was it good until the environment (external) changed? And can we distinguish between the two without checking every memory?***

In this paper, we propose Bayesian-Harness, a Bayesian framework that models harness memory as a generative process. Rather than assigning a single value to score each memory item, as in previous approaches, we define two latent random variables, *quality* and *validity*, for each memory item (Figure 1). Specifically, the validity, v_k, is a binary latent state that fails at a Markov-modulated hazard and is restored only by a rewrite. A regime variable z_g controls the hazard for each subsystem. It is a two-state Markov chain (Rabiner, 1989) that determines when the subsystem enters and leaves a period of change. This gives a simple chain, $z_g \to v_k \to y$. The regime affects whether a memory item is still valid, which then affects whether the task succeeds. Quality determines how likely the task is to succeed when the memory is still valid. For instance, for an item about a method in an API, we can ask whether the captured information is of sufficient quality (e.g., parameters given with types, effects of method correct) or whether the item becomes completely invalid if the method was moved to a different API.

To investigate the effectiveness of our method in harness memory management, we build two datasets, MASKEDMIG and ACMESDK. MASKEDMIG consists of 157 masked API migrations mined from 18 real Python libraries, which anchors the method in real drift, and ACMESDK is a generated company SDK whose 36 monthly releases follow real version history, which gives us the controlled drift that real libraries cannot provide (Jain et al., 2025) (Appendix A.3). Across open-weight models from 4B to 27B and two closed-weight Gemini models, our approach Bayesian-Harness is the best maintenance policy on every model of MASKEDMIG, 3.1–8.3 points above the strongest baseline of each and above Bayesian-Agent (Wu et al., 2026) everywhere, and in all 15 model–budget configurations of ACMESDK, including stress tests that depart from its structural assumptions (Appendix A.8).

This paper makes the following main contributions:

- We model LLM-agent knowledge staleness as a Bayesian generative process with two latent variables per memory: validity, which forms a Markov-modulated chain $z_g \to v_k \to y$, and quality, which has a hierarchical Beta prior shared within each subsystem.
- We build two datasets, MASKEDMIG and ACMESDK. MASKEDMIG is built from version releases of 18 real Python libraries, while ACMESDK is based on SOP-Bench (Nandi et al., 2026) and imitates real company SDK change histories. We release both datasets on a public platform for non-commercial research use.
- Across five LLMs of different capacities, open-weight Qwen3.5 models from 4B to 27B and the closed-weight Gemini 2.5 Flash-Lite and 3.1 Pro, we show how Bayesian-Harness improves an agentic system by 3.1–8.3 points over the strongest baseline of each on the mined migrations of MASKEDMIG, with gains that persist under structural stress tests on ACMESDK.

2 Related Work

Most methods that manage the knowledge an agent accumulates across episodes assign a single value to each *memory item* (also *knowledge item*, *memory*, or just *item*) and use a threshold to decide what to update or delete. They fall into three families. Capacity-driven methods score what to keep and compress the rest when the context is full (Packer et al., 2023; Chhikara et al., 2025; Xu et al., 2025; Li et al., 2025; Rasmussen et al., 2025). Time-driven methods let memories decay and retire those untouched for long enough (Park et al., 2023; Zhong et al., 2024; Lam et al., 2026; Ji et al., 2026). Outcome-driven methods keep a running score per episode, keeping what succeeds and retiring what fails (Wang et al., 2024a; Shinn et al., 2023; Zhao et al., 2024; Wang et al., 2025b); recent skill libraries add explicit curation of that experience (Suzgun et al., 2026; Zheng et al., 2025a; Ouyang et al., 2026b; Zhang et al., 2026; Fang et al., 2026; Ouyang et al., 2026a; Xia et al., 2026), and a few keep probabilistic beliefs rather than scores (Forouzandeh et al., 2026; Liao et al., 2026), though these beliefs concern only action reliability or hidden state, not when an item stops being valid. Surveys (Zhang et al., 2025; Luo et al., 2026; Ding et al., 2026) organise this literature by what is stored and how it is retrieved and updated. None of them separates what the agent itself does to a memory item from what the external environment does to it.

Bayesian-Agent (Wu et al., 2026) is the closest work to ours. It keeps a feature-conditioned categorical posterior over the outcomes of each skill and maps it to actions such as patch, split, compress, retire and explore: it patches a skill when the same failure mode recurs and retires it when its success posterior falls. We share its Bayesian framing, but differ in separating validity from quality, explicitly modelling time and change, and spending an explicit verification budget. STALE, Supersede and EvoArena (Chao et al., 2026; Patel, 2026; Xu et al., 2026), building on LongMemEval (Wu et al., 2025), test whether an agent revises conversational memories that later evidence has invalidated; we instead maintain memories about executable interfaces under a budget.

Benchmarks such as SWE-bench (Jimenez et al., 2024), τ-bench (Yao et al., 2025), SOP-Bench (Nandi et al., 2026), LifelongAgentBench (Zheng et al., 2025b), Long-Horizon-Terminal-Bench (Li et al., 2026), and BigCodeBench (Zhuo et al., 2025) test agents on real repositories, procedures, long-horizon tasks, and real libraries. However, evaluating on real libraries can confound the value of a memory item with knowledge of library history acquired during pre-training (Xu et al., 2024a). To isolate this effect, our controlled ACMESDK benchmark uses realistic release histories, following the spirit of procedurally generated environments (Cobbe et al., 2020) and time-stamped benchmarks designed to reduce pre-training exposure (Jain et al., 2025).

The need to track changing library knowledge is itself well established. APIs are deprecated and undergo breaking changes at a steady rate (Dig & Johnson, 2006; Robbes et al., 2012; Wang et al., 2020; Cogo et al., 2022; Ochoa et al., 2022), developers lag in adopting replacements (Kula et al., 2018), and code LLMs continue to emit deprecated calls after cutoff (Wang et al., 2025a; Ashik et al., 2026). Version-aware code benchmarks (Wu et al., 2024; Kuhar et al., 2025; Misra et al., 2026) measure this problem on real libraries, while knowledge editing and unlearning methods address it by modifying model weights (Liu et al., 2024; Tran et al., 2026), alongside broader work on stale facts in models (Kasai et al., 2023; Cheng et al., 2024; Xu et al., 2024b). Our setting is different: we maintain memories outside the model and repair them by re-verifying and rewriting. In MASKEDMIG, we further mask successor names so that pre-training cannot supply them.

3 Methodology

3.1 Problem formulation

An agentic harness accumulates reusable knowledge items $K = \{k_1, \ldots, k_N\}$, including memory entries, tool-usage information, skills, and procedures. An item can fail for two different reasons: it has become stale because the environment changed, or it is unreliable even while it is valid. These two cases should not be treated in the same way, so we maintain two latent variables for each item k. Its validity is

$$v_k \in \{0, 1\}, \qquad s_k := \Pr(v_k = 0), \tag{1}$$

where $v_k = 1$ means that the knowledge still applies and s_k is its staleness probability. Its quality is

$$p_k \in [0,1], \qquad p_k = \Pr(y = 1 \mid v_k = 1), \tag{2}$$

the probability that the item works when it is valid. Validity can change quickly with the environment, while quality changes gradually with experience.

Items are grouped by subsystem $g(k)$, and a subsystem has a latent regime

$$z_g \in \{\text{STABLE}, \text{CHANGING}\}, \qquad b_g := \Pr(z_g = \text{CHANGING}), \tag{3}$$

which captures periods in which many items of one subsystem change at once. With $y \in \{0,1\}$ the outcome when the item is used, the dependency is

$$z_g \longrightarrow v_k \longrightarrow y, \qquad p_k \longrightarrow y. \tag{4}$$

This separation is the central idea of Bayesian-Harness: a failure can be evidence that the world changed without being evidence that the knowledge itself is of low quality.

3.2 Bayesian state model

Time is measured in maintenance periods, one release (a month in ACMESDK); the regime moves in continuous time and validity in discrete time, both on this per-period clock.

Regime dynamics. Each subsystem regime is a two-state continuous-time Markov chain: a stable subsystem enters CHANGING at rate λ and returns to STABLE at rate μ. Over a gap of Δt periods the belief in the changing regime is

$$b_g^- = \pi_\infty + (b_g - \pi_\infty)e^{-(\lambda+\mu)\Delta t}, \qquad \pi_\infty = \frac{\lambda}{\lambda+\mu}, \tag{5}$$

so a subsystem can recover after a month of heavy change instead of growing riskier monotonically with age. We fix μ and learn λ online with its Gamma–Poisson conjugate prior (Appendix A.1).

Validity dynamics. Conditional on the regime, a valid item becomes stale within one period with probability

$$h_g = \begin{cases} \varepsilon, & z_g = \text{STABLE}, \\ \varepsilon + \rho, & z_g = \text{CHANGING}, \end{cases} \tag{6}$$

and staleness is absorbing until the item is rewritten, so $s_k^-(z_g) = s_k + (1 - s_k)h_g$. Marginalising the unknown regime gives the predictive staleness probability

$$\bar{s}_k^- = (1 - b_g^-)\, s_k^-(\text{STABLE}) + b_g^-\, s_k^-(\text{CHANGING}) = s_k + (1 - s_k)(\varepsilon + \rho\, b_g^-), \tag{7}$$

composed Δt times over longer gaps with the regime belief advanced alongside it. Time alone raises the probability that an item is stale; the regime belief sets how fast. The values of ε, ρ and μ are properties of library drift in general, measured on public release histories and not on any evaluation world (Appendix A.2).

3.3 Observation and Bayesian update

When item k is used, the observed outcome follows

$$\Pr(y = 1 \mid v_k = 1, p_k) = p_k, \qquad \Pr(y = 1 \mid v_k = 0) = q, \tag{8}$$

where q is small because stale knowledge occasionally succeeds by chance, and the regime affects the outcome only through validity, $\Pr(y \mid v_k, z_g) = \Pr(y \mid v_k)$. Since v_k and z_g are both binary, the posterior over their four joint configurations is exact (Appendix A.1). For item k, it reduces to the two-hypothesis update, in odds form,

$$\frac{s_k^+}{1 - s_k^+} = \frac{\bar{s}_k^-}{1 - \bar{s}_k^-} \frac{L(y \mid v_k = 0)}{L(y \mid v_k = 1)}, \qquad \frac{L(y = 0 \mid v_k = 0)}{L(y = 0 \mid v_k = 1)} = \frac{1 - q}{1 - \bar{p}_k}, \tag{9}$$

where $\bar{p}_k$ is the posterior mean quality defined in Eq. 11 (§3.4), so a failure is stronger evidence of staleness for a historically reliable item than for one that has often failed. For the regime,

$$\frac{b_g^+}{1 - b_g^+} = \frac{b_g^-}{1 - b_g^-} \frac{s_k^-(\text{CHANGING})L(y \mid 0) + [1 - s_k^-(\text{CHANGING})]L(y \mid 1)}{s_k^-(\text{STABLE})L(y \mid 0) + [1 - s_k^-(\text{STABLE})]L(y \mid 1)}, \tag{10}$$

so one failure can raise the estimated probability that the whole subsystem is changing, and with it the staleness beliefs of its other items.

Two further evidence channels use the same update. A classified error message provides noisy evidence about whether an item is stale. An active verification runs the item's stored example against the current environment and observes its validity directly, with likelihood 1 for the observed state and 0 for the other; it certifies the example, not the prose around it. When several items are in context and the outcome cannot be attributed to one of them, the evidence is shared equally (Appendix A.1).

3.4 Quality estimation

Quality is the success rate of an item while it is valid, modelled by a hierarchical Beta distribution,

$$p_k \sim \mathrm{Beta}\big(n_k^{\mathrm{s}} + \kappa m_{g(k)},\ n_k^{\mathrm{f}} + \kappa[1 - m_{g(k)}]\big)\,, \qquad \bar{p}_k = \mathbb{E}[p_k] = \frac{n_k^{\mathrm{s}} + \kappa m_{g(k)}}{n_k^{\mathrm{s}} + n_k^{\mathrm{f}} + \kappa}, \tag{11}$$

where n_k^{s} and n_k^{f} are the item's effective success and failure counts, m_g is the mean quality of its subsystem, and κ is the number of pseudo-observations borrowed from it. The subsystem mean has its own $\mathrm{Beta}(\alpha_0, \beta_0)$ prior and pools the counts of its items,

$$m_g = \frac{\alpha_0 + \sum_{k:g(k)=g} n_k^{\mathrm{s}}}{\alpha_0 + \beta_0 + \sum_{k:g(k)=g}(n_k^{\mathrm{s}} + n_k^{\mathrm{f}})}, \tag{12}$$

which is standard three-level partial pooling (Gelman et al., 2013): an item with a long record is dominated by its own counts, a short record falls back on its subsystem, and a young subsystem on the shared prior.

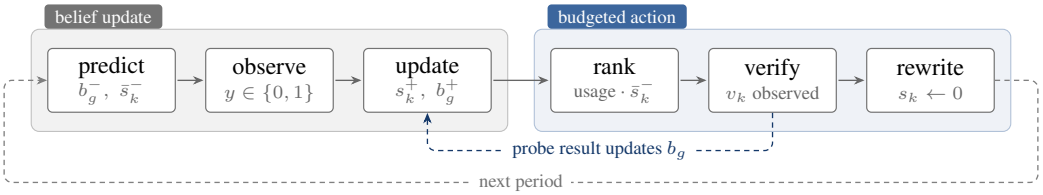
Because a failure may come from an item that is already stale, the counts are updated with the posterior responsibility $w_k := \Pr(v_k = 1 \mid y) = 1 - s_k^+$,

$$n_k^{\mathrm{s}} \leftarrow n_k^{\mathrm{s}} + w_k \quad (y = 1), \qquad n_k^{\mathrm{f}} \leftarrow n_k^{\mathrm{f}} + w_k \quad (y = 0), \tag{13}$$

an EM-style expected-count update: a failure attributed mainly to the environment barely moves quality, while one attributed to a valid item lowers it (Appendix A.1 discusses the approximations involved). When an item is rewritten, its validity is reset, $s_k \leftarrow 0$, while $(n_k^{\mathrm{s}}, n_k^{\mathrm{f}})$ are kept, so the new memory inherits the record of its predecessor but not its stale-state belief.

3.5 Maintenance policy

Each period the harness spends a fixed budget on probes and rewrites. It ranks items by staleness $\bar{s}_k^-$ weighted by observed usage and verifies them in that order. A probe also moves the shared regime belief b_g, and with it the priority of the other items in the subsystem from the next period on. A verified-stale item is rewritten at once; if the remaining budget cannot pay for the rewrite, the rewrite is deferred until the budget allows. No hand-set failure threshold enters any decision. The loop of one period is:



4 Experiments

We evaluate on two complementary corpora that capture API changes over time. These changes require the agent to adapt and may affect the validity and quality of retained memory items. The two corpora are MASKEDMIG, which contains 157 migrations mined from real library histories, and ACMESDK, a newly generated company-style SDK whose drift is calibrated using real repository histories (Appendix A.3).

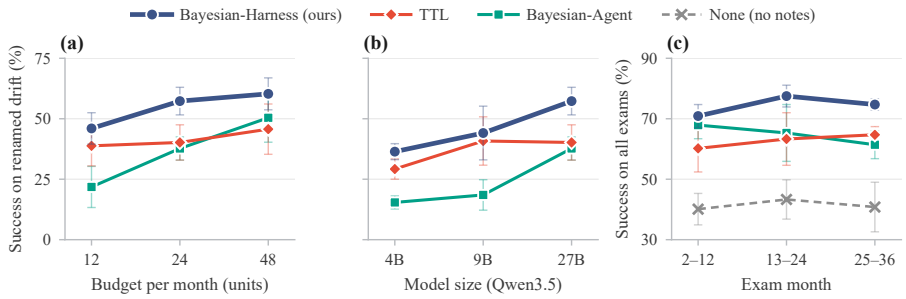


Figure 2: **ACMESDK results by budget, model size and time.** Monthly test success (↑), three seeds pooled, bars are the s.d. across seeds; Qwen3.5-27B at budget 24 unless the axis varies it. (a, b) Renamed drift against budget and model size; (c) all monthly tests by period. Full tables in Appendix A.5.

4.1 THE MASKEDMIG CORPUS

MASKEDMIG collects 157 API migrations from 18 Python libraries in which, in a release between April 2025 and August 2026, the maintainers removed an API and named a successor. The agent starts with one memory item per migration written against the old version; when the new release arrives, the items are stale, and the agent spends a limited budget verifying and rewriting them before the monthly test.

Unmasked real-library evaluation can conflate maintenance with release-history recall: an agent with no memory items repairs 60–78% of the broken snippets. We therefore replace every successor name with a random hash in the package, the changelog, and the monthly test; a monthly test passes only if the frozen assertions hold and the masked successor is called (Appendix A.3).

4.2 THE ACMESDK CORPUS

ACMESDK is a controlled company-style Python SDK with 12 subsystems and 412–417 public APIs, each with deterministic, machine-checkable behaviour. We withhold documentation and introspection so that performance reflects knowledge maintained in the harness (Appendix A.3).

ACMESDK has 36 monthly releases with five types of changes, grouped into three categories in the tables: renames and removals with successors (*renamed* in the tables), parameter changes (*param*), and return-shape and semantic changes (*behav.*), generated as a background trickle, subsystem-wide heavy months and package-wide convention changes calibrated on real repositories. Each release ships a changelog; a repair may read its subsystem's entries only after selecting a memory item, so the changelog never tells a policy which item to spend its budget on, and the test step never sees it.

At month 1 the agent holds one verified memory item (capturing usage) per API. In each monthly test it receives a broken snippet, may issue up to three shell commands, and must repair it while preserving the original `assert` lines, calling the current API, and exiting cleanly. Up to 10 tests a month are randomly selected from broken APIs, and three seeds give independent worlds.

4.3 SETUP

Each month the harness updates its beliefs from the previous tests, spends a maintenance budget (24 units unless stated otherwise) on probes and repairs, and then evaluates the monthly tests. A probe checks whether a memory item's snippet still runs; a repair lets the model rewrite a broken item from the changelog and counts only if the new snippet passes the original assertion. At test time, the agent sees the memory item for the tested API and its two most-used memories from its subsystem.

None keeps no memory items and *Static* never maintains them; *Random* probes items at random, *TTL* probes the least recently verified item, and *Bayesian-Agent* (Wu et al., 2026) keeps a feature-conditioned categorical posterior per item; our memory items carry no context features, so its feature-free success posterior reduces to the mean of a per-item Beta posterior. Bayesian-Harness uses the model of §3 with the 12 subsystems as groups; both Bayesian methods rank items by staleness and usage, which for Bayesian-Agent is one minus its posterior mean. The constants of

Table 1: **Main result. Monthly test success (↑) on MASKEDMIG, 157 real API migrations with masked successor names**, budget 24 per quarter, library as the shared group. Seeds differ only in the quarter in which each monthly test is met; ± is the s.d. across three seeds. Bold marks the best maintenance policy.

Method	Qwen3.5-4B	Qwen3.5-9B	Qwen3.5-27B	Gemini 2.5 Flash-Lite	Gemini 3.1 Pro
Oracle	27.4±1.9	38.9±2.5	52.2±1.3	22.3±1.9	65.6±3.2
None	0.6±0.6	0.6±1.1	12.1±0.6	0.0±0.0	12.7±3.2
Static	0.6±0.0	1.3±2.2	8.9±8.3	0.6±1.1	11.5±2.8
Random	16.6±5.1	21.0±8.0	37.6±1.7	8.9±5.0	36.9±4.2
Bayesian-Agent	14.0±4.8	18.5±2.3	36.3±7.8	9.6±4.2	53.5±3.9
TTL	17.8±3.9	22.9±2.3	40.1±7.6	8.9±1.7	51.0±2.8
Bayesian-Harness	**25.5**±1.7	**29.9**±2.2	**48.4**±2.8	**12.7**±1.3	**58.6**±2.2

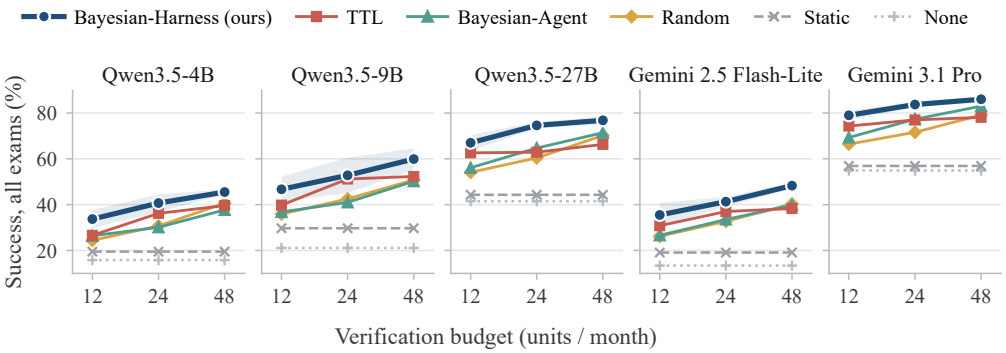


Figure 3: **Monthly test success (↑) on ACMESDK against the monthly budget.** Success on all tests, one panel per model; points pool three seeds and the band is the s.d. of Bayesian-Harness across seeds. The numbers, with Oracle, are in Table 6 (Appendix A.5).

Bayesian-Harness are fixed from disjoint public release histories (Table 5, Appendix A.2). *Oracle* observes which items are stale, but not which repairs the model can complete, and repairs stale items in order of use; it is therefore a diagnostic reference rather than a performance ceiling. Models are Qwen3.5 at 4B, 9B and 27B (Yang et al., 2025) and Gemini 2.5 Flash-Lite and 3.1 Pro (Comanici et al., 2025), run at temperature 0 with three seeds per configuration unless a caption says otherwise; cells pool the seeds and give the s.d. after ±.

4.4 MAIN RESULTS

Our main results are shown in Table 1. The 157 migrations of MASKEDMIG are changes real maintainers shipped on real libraries, and the masked successor names keep the monthly test from being passed from memory of the release. Figure 3 and Table 6 (Appendix A.5) then repeat the comparison on ACMESDK, where we generate the drift instead of mining it and can therefore move the budget, the model size and the type of change one at a time.

On the 157 masked real migrations (Table 1), Bayesian-Harness is the best maintenance policy in every column: 7.0–8.3 points above TTL and 11.4–12.1 above Bayesian-Agent at the three Qwen3.5 sizes, and 5.1 above Bayesian-Agent with Gemini 3.1 Pro. TTL is the strongest baseline at the Qwen3.5 sizes, as it is in most ACMESDK settings (Table 6). All policies have trouble achieving test success with Flash-Lite. Even the oracle only achieves 22.3%; Bayesian-Harness leads the strongest baseline, Bayesian-Agent, by 3.1 points. ACMESDK asks the same question under drift we control (Table 6), over three budgets and five models.

Bayesian-Harness is the best maintenance policy in every cell of Table 6, 17.9–38.8 points above *None* (Δ rows). At 12 units, it is 0.4 points above Oracle on average and ahead of it in three of the five models; Oracle leads by 3.7 points at 24 units and 8.1 at 48. The oracle knows which items are stale but not which repairs the model can complete, so it retries a failed repair of a much-used item every month; at 12 units (four repairs) such items take much of its budget. Bayesian-Harness, like the other policies, resets an item's staleness after a repair attempt and moves on, so an impossible repair costs it one attempt. With more budget, Oracle's exact knowledge of staleness wins.

With Qwen3.5-27B, Bayesian-Harness is the best policy on renamed and on parameter drift in Table 7; behaviour drift is near-saturated at 96–99%, so renamed and parameter drift provide the more discriminative comparisons. On renamed drift, it reaches 57.3%, outperforming Bayesian-Agent by 19.6 points, TTL by 17.1, and Random by 28.9, and ahead of Bayesian-Agent and TTL in each of the three seeds (Appendix A.5). Across all 1,019 monthly tests, Bayesian-Harness reaches 74.6%, compared with 64.7% for Bayesian-Agent, 62.9% for TTL, and 60.4% for Random. The gap is smaller on parameter drift, where the old call shape already provides useful information. Bayesian-Harness is also closest to the oracle, only 5.8 points behind on renamed drift and 3.5 points overall.

How the gap to oracle changes over the 36 months, and how the two Gemini models behave, are reported in Appendix A.5. Bayesian-Harness also keeps its lead under stress tests that depart from the model's structural assumptions (Table 4, Appendix A.8; command-line memories in Appendix A.11). With every memory item filed under a wrong subsystem, it still reaches 50.1 / 69.0, above TTL and Bayesian-Agent in the unperturbed world, and when bursts follow a hidden axis instead of the subsystems, it leads Bayesian-Agent by 12.6 points on renamed drift.

4.5 Ablations and sensitivity

This section asks which components of Bayesian-Harness contribute to the improvement. Table 2 switches off one part at a time and keeps the rest on. Recovery carries the largest single effect on renamed drift. Forcing a subsystem to stay in CHANGING ($\mu \to 0$) costs 8.0 points there and 4.6 overall. The two sharing mechanisms are the ones that matter most across all monthly tests. Removing the shared regime costs 6.6 / 5.7 and removing pooling costs 5.2 / 5.1, since both share evidence across the memory items of a subsystem, allowing them to cover tests of every drift type. The error channel helps by the smallest margin on both, 4.1 / 3.8. No component is worth less than 3.8 points overall, so every part contributes to the model. The same pattern on MASKEDMIG is reported in Appendix A.7.

Table 2: **Ablation of Bayesian-Harness**, Qwen3.5-27B, budget 24, three seeds, one component off per row. Cells show test success on renamed drift / all monthly tests (↑), ± s.d. across seeds. Recovery (μ) lets a subsystem turn stable again; the shared regime z_g pools staleness and pooling (κ) pools quality within a subsystem; the error channel reads failed calls as evidence of staleness.

	renamed / all
Bayesian-Harness	**57.3**±5.7 / **74.6**±1.4
no recovery	49.3±3.7 / 70.0±2.8
no error channel	53.2±6.9 / 70.8±2.1
no pooling	52.1±5.0 / 69.5±3.9
no shared regime	50.7±5.4 / 68.9±2.4

Figure 2(a) and (b) vary the budget and the model size on renamed drift, with the full numbers in Table 9 (Appendix A.5). Bayesian-Harness is the best policy in every cell (budget efficiency in Appendix A.9). A protocol variant with cheaper mechanical repairs is reported in Appendix A.5.

4.6 Can the model tell a low-quality item from a stale one?

The experiments above test staleness only. To test whether Bayesian-Harness also separates a low-quality item from a stale one, we assign a random half of the functions in ACMESDK, and half of the migrations in MASKEDMIG, to *cheap* repairs whose item text no longer helps the test agent, and label an item *stale* if its interface changed after its last repair and *low-quality* if its last repair was cheap. For each policy we report the ranking score it gives to low-quality but valid and to high-quality but stale items, its quality estimate on cheaply and fully repaired valid items, the share of probes spent on valid memories, the share of stale items caught within one period, and monthly test success. Besides Bayesian-Agent we compare *two counters* (a TTL clock plus a running success rate), Bayesian-Harness with quality frozen at $\bar{p}_k = 1$, and a variant of Bayesian-Harness that also rewrites low-quality items. Appendices A.10 and A.6 give the protocol and more ablations.

On ACMESDK with Qwen3.5-27B, Bayesian-Harness scores low-quality but valid items at 0.16 and good but stale items at 0.39 at budget 12, and at 0.16 and 0.37 at budget 24. Bayesian-Agent scores the two groups at 0.42 and 0.50; with quality frozen, they are 0.59 and 0.84, and false alarms rise from 62.8% to 81.0% of probes at budget 12. Renamed-drift success at budget 12 is 38.0% for Bayesian-Harness, 29.5% for two counters, 26.7% with quality frozen and 22.0% for Bayesian-Agent; at budget 24 the four are 49.0, 44.1, 45.5 and 30.6, and for Qwen3.5-9B at budget 12 they are 24.8, 14.9, 13.8 and 9.1. Rewriting on low quality lowers renamed success on ACMESDK by

Table 3: **Low-quality items against stale items**, three seeds, ± s.d. Score is the ranking quantity (for two counters, periods since verification plus alarms); quality is $\bar{p}_k$; false alarms are probes on valid items; caught is broken items verified within one period. While valid, cheap / full repairs pass 85.1 / 100.0% of monthly tests on ACMESDK and 11.0 / 100.0% on MASKEDMIG.

	Score on		Quality on valid memories		Budget (%)		Monthly test success (%)	
Policy	low-quality, valid	good, stale	cheap repair	full repair	false alarms	caught	renamed	all
ACMESDK, *Qwen3.5-27B, budget 12 per month*								
Bayesian-Agent (single score)	0.42±0.06	0.50±0.01	n/a	n/a	76.2±10.5	3.2±1.2	22.0±6.2	53.5±4.1
Two counters (TTL clock + success rate)	5.6±0.7	20.0±1.1	0.58±0.05	0.61±0.01	79.2±5.0	3.7±1.1	29.5±8.3	56.0±4.1
Bayesian-Harness, quality frozen ($\bar{p}_k$=1)	0.59±0.11	0.84±0.04	n/a	n/a	81.0±5.5	5.2±1.1	26.7±3.7	55.5±0.9
Bayesian-Harness	0.16±0.03	0.39±0.07	0.57±0.04	0.68±0.04	62.8±5.7	3.7±1.1	38.0±8.7	61.3±2.6
Bayesian-Harness + rewrite on low quality	0.19±0.07	0.42±0.07	0.62±0.04	0.66±0.02	66.1±6.0	4.0±1.3	31.7±6.1	57.7±2.0
ACMESDK, *Qwen3.5-27B, budget 24 per month*								
Bayesian-Agent (single score)	0.41±0.02	0.49±0.01	n/a	n/a	75.8±3.6	6.0±1.9	30.6±7.1	57.8±2.4
Two counters (TTL clock + success rate)	5.6±1.3	19.2±1.2	0.57±0.02	0.58±0.00	71.2±6.0	5.6±1.8	44.1±9.5	64.0±6.3
Bayesian-Harness, quality frozen ($\bar{p}_k$=1)	0.41±0.05	0.78±0.03	n/a	n/a	77.7±4.7	7.9±2.3	45.5±10.4	64.5±5.9
Bayesian-Harness	0.16±0.02	0.37±0.02	0.62±0.04	0.73±0.04	68.7±4.0	7.4±0.9	49.0±6.8	66.2±3.3
Bayesian-Harness + rewrite on low quality	0.16±0.03	0.37±0.03	0.70±0.03	0.71±0.03	66.1±3.5	5.9±0.8	46.3±7.2	64.4±3.9
ACMESDK, *Qwen3.5-9B, budget 12 per month*								
Bayesian-Agent (single score)	0.55±0.07	0.52±0.00	n/a	n/a	68.7±10.6	2.2±0.7	9.1±1.7	34.3±2.1
Two counters (TTL clock + success rate)	7.4±1.7	21.5±0.8	0.56±0.05	0.63±0.01	80.8±8.4	3.3±1.0	14.9±4.0	37.7±2.0
Bayesian-Harness, quality frozen ($\bar{p}_k$=1)	0.64±0.12	0.88±0.03	n/a	n/a	77.0±10.3	3.9±1.2	13.8±2.0	35.5±1.8
Bayesian-Harness	0.31±0.14	0.50±0.05	0.51±0.09	0.61±0.06	62.3±8.6	5.4±0.5	24.8±7.5	42.7±2.1
Bayesian-Harness + rewrite on low quality	0.38±0.22	0.50±0.04	0.50±0.06	0.58±0.01	61.6±9.7	3.9±1.2	19.6±3.5	39.4±3.3
MASKEDMIG, *Qwen3.5-27B, budget 24 per quarter, 471 monthly tests per seed*								
Bayesian-Agent (single score)	0.63±0.03	0.54±0.01	n/a	n/a	68.4±1.5	8.2±1.1	24.1±3.1	25.2±2.1
Two counters (TTL clock + success rate)	28.1±4.9	23.4±2.5	0.53±0.02	0.64±0.01	53.8±2.0	9.3±0.4	25.4±3.5	27.2±2.1
Bayesian-Harness, quality frozen ($\bar{p}_k$=1)	0.84±0.04	0.80±0.01	n/a	n/a	66.9±1.6	9.7±0.7	23.2±4.9	25.6±1.9
Bayesian-Harness	0.50±0.12	0.67±0.03	0.33±0.04	0.43±0.05	61.3±2.5	7.4±0.4	35.4±2.8	38.4±3.3
Bayesian-Harness + rewrite on low quality	0.38±0.09	0.67±0.03	0.48±0.04	0.57±0.04	42.9±5.9	7.0±0.0	36.1±3.1	39.3±2.5

6.3, 2.7 and 5.2 points. On MASKEDMIG, a cheaply repaired item passes 11.0% of monthly tests while valid, against 100% after a full repair. The three other policies that keep such items score 25.2–27.2% on all monthly tests, against 38.4% for Bayesian-Harness. Bayesian-Harness scores low-quality but valid items at 0.50 and good but stale items at 0.67, with quality estimates of 0.33 for cheap and 0.43 for full repairs; Bayesian-Agent scores them at 0.63 and 0.54, the wrong way round. The rewrite variant, which replaces the cheap items it detects, reaches 36.1% on renamed and 39.3% on all monthly tests, within one s.d., and cuts false alarms from 61.3% to 42.9%.

Table 4: **Robustness to model and world mismatch.** Success on renamed drift / all monthly tests (↑), Qwen3.5-27B, budget 24, three seeds per row, ± s.d. across seeds. Misfiled rows file memory items under a wrong subsystem; the baselines and oracle use no groups and are unaffected (—). ε_w is the background break rate per function and month. Bold marks the best maintenance policy.

World	Bayesian-Harness	TTL	Bayesian-Agent	Oracle
evaluation world (6 subsystem-wide heavy months, $\varepsilon_w = 0.010$)	**57.3**±5.7 / **74.6**±1.4	40.2±7.3 / 62.9±4.4	37.7±4.8 / 64.7±3.8	63.1±9.2 / 78.1±3.2
wrong groups				
30% of memories filed under a wrong subsystem	**54.0**±8.6 / **70.2**±2.6	—	—	—
every memory filed under a wrong subsystem	**50.1**±8.5 / **69.0**±3.2	—	—	—
different drift rates				
no heavy months, $\varepsilon_w = 0.024$	**50.6**±1.3 / **74.8**±3.1	32.9±4.2 / 66.0±4.2	33.3±5.8 / 66.6±5.3	63.0±9.6 / 81.0±6.0
11 heavy months, $\varepsilon_w = 0.005$	**54.1**±14.4 / **72.8**±5.0	42.4±12.9 / 64.5±5.1	36.6±12.1 / 62.0±3.4	69.0±4.2 / 80.8±1.0
different structure				
bursts on a hidden axis, not on subsystems	**59.1**±5.0 / **85.4**±2.2	40.9±12.0 / 80.8±4.0	46.5±3.4 / 80.2±5.7	70.9±11.9 / 88.5±4.6
heavy months on a fixed 5-month schedule	**51.6**±5.1 / **72.2**±4.0	32.0±3.5 / 58.5±4.5	37.1±8.6 / 61.0±5.6	64.8±6.1 / 79.7±3.7
heavy periods of 6 and 3 months, not 1	**52.8**±8.4 / **70.9**±1.9	39.1±8.8 / 60.0±4.4	38.1±14.0 / 58.9±5.5	62.2±7.5 / 75.6±1.6

5 CONCLUSION

An LLM-agent memory item can fail because the environment changed (external) or because it was of low quality (internal), yet existing approaches keep a single value for each piece of knowledge. Instead, Bayesian-Harness separates validity from quality, estimates both from monthly test outcomes and cheap probes, detects when a period of heavy change starts and ends, and spends the budget on the memory items most likely to be stale. On 157 masked migrations extracted from real library histories, Bayesian-Harness is the best maintenance policy for every model we test, 3.1–8.3 points above the strongest baseline of each. On our contamination-resistant controlled benchmark, it is the best in all 15 model–budget configurations and remains best under structural stress tests.

AI USE STATEMENT

In this work, we used generative AI tools to build a synthetic dataset. The ACMESDK corpus, its 12 subsystems, its 412–417 API behaviours, the memory item with usage notes attached to each API, and the changelog of each release, were written by Claude Code (Fable 5.1), seeded from the enterprise subsystems of SOP-Bench (Nandi et al., 2026). We specified the drift rules, the change types and their weights, the three layers that place the changes, and the maintenance and test protocol. Claude Code (Fable 5.1) then wrote the package, its 13 kernel templates and the release history generator that realise them, as Appendix A.3 describes. We also used generative AI tools for polishing sentences. We have not used generative AI tools to propose or refine the research hypothesis, to design the protocol or the experiments, to develop the probabilistic model of §3, to formulate or prove its mathematical claims, or to interpret the experimental results. We have reviewed all AI-assisted work. Every generated API has a deterministic behaviour that the harness checks by execution, so a generated memory item is only accepted once its snippet runs and its assertions hold; two authors additionally read a random sample of 53 memory items and 48 changelog entries. We take responsibility for the final content of this work, including text, claims or artefacts produced with the aid of generative AI.

ETHICS STATEMENT

This work involves no human subjects, no personal data and no user study. The ACMESDK corpus is synthetic, but it is not built from nothing. Its subsystem layout and its business vocabulary are seeded from SOP-Bench (Nandi et al., 2026), whose data set is released under CC BY-NC 4.0. We take that seed only. No SOP-Bench task, tool implementation or trajectory is reused, and Appendix A.3 states what the seed contributed and what was generated from it. We attribute SOP-Bench wherever ACMESDK is described, and we release ACMESDK for non-commercial research use. MASKEDMIG is built from the public release histories of 18 open-source Python libraries and records only public API names, public changelog entries and public release dates; we redistribute no library code. The method maintains an agent's own memory items about the software it calls and is not intended to act on people or on their data. Its main foreseeable misuse is the same as that of any agent that edits code, so the harness never executes a repair outside a sandbox that is pinned to one release directory. We release the corpora and the harness so that the results can be checked (Appendix A.3).

REPRODUCIBILITY STATEMENT

The model has no parameter that was tuned on an evaluation world. Appendix A.2 lists every prior and constant with the reason it was set and, for the dynamics constants, the public release histories they were measured on. Appendix A.3 gives the world generator, with the change types and their weights, the three layers that place the changes, and the release-history statistics we calibrated them against. Appendix A.4 gives the monthly test prompt and the repair prompt, which are held fixed across every method. Configurations are run at temperature 0 with three seeds unless the table says otherwise; Appendix A.5 reports each seed on its own at budget 24, and every row of Table 4 pools three seeds. The code is available at `github.com/anonymouspaper987/bayesian_harness`, and the generated worlds and the monthly test sets at `github.com/anonymouspaper987/bayesian_harness_datasets`.

REFERENCES

Ahmed Nusayer Ashik, Shaowei Wang, Tse-Hsun Chen, Muhammad Asaduzzaman, and Yuan Tian. When LLMs lag behind: Knowledge conflicts from evolving APIs in code generation. arXiv preprint arXiv:2604.09515, April 2026.

Hanxiang Chao, Yihan Bai, Rui Sheng, Tianle Li, and Yushi Sun. STALE: Can LLM agents know when their memories are no longer valid? arXiv preprint arXiv:2605.06527, May 2026.

Jeffrey Cheng, Marc Marone, Orion Weller, Dawn Lawrie, Daniel Khashabi, and Benjamin Van Durme. Dated data: Tracing knowledge cutoffs in large language models. In *Conference on Language Modeling*, 2024.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready AI agents with scalable long-term memory. In *European Conference on Artificial Intelligence*, Frontiers in Artificial Intelligence and Applications. IOS Press, 2025. doi: 10.3233/FAIA251160.

Karl Cobbe, Chris Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning. In *International Conference on Machine Learning*, pp. 2048–2056. PMLR, 2020.

Filipe R. Cogo, Gustavo A. Oliva, and Ahmed E. Hassan. Deprecation of packages and releases in software ecosystems: A case study on npm. *IEEE Transactions on Software Engineering*, 48(7): 2208–2223, 2022. doi: 10.1109/TSE.2021.3055123.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. arXiv preprint arXiv:2507.06261, July 2025.

Danny Dig and Ralph Johnson. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, March 2006. doi: 10.1002/smr.328.

Tianyu Ding, Aditya Nannapaneni, Bingfan Liu, and Ling Zhang. Always-on agents: A survey of persistent memory, state, and governance in LLM agents. arXiv preprint arXiv:2606.30306, June 2026.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Memp: Exploring agent procedural memory. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 17490–17502, 2026.

Saman Forouzandeh, Wei Peng, Parham Moradi, Xinghuo Yu, and Mahdi Jalili. Learning hierarchical procedural memory for LLM agents through Bayesian selection and contrastive refinement. In *Proceedings of the 25th International Conference on Autonomous Agents and Multiagent Systems*, 2026. doi: 10.65109/FKYO8341.

Andrew Gelman, John B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin. *Bayesian data analysis*. Chapman and Hall/CRC, 3rd edition, 2013. doi: 10.1201/b16018.

Gonzalo Gonzalez-Pumariega, Saaket Agashe, Jiachen Yang, Ang Li, and Xin Eric Wang. On the reliability of computer use agents. arXiv preprint arXiv:2604.17849, April 2026.

Hechuan Guo, Yongle Hao, Yue Zhang, Minghui Xu, Peizhuo Lv, Jiezhi Chen, and Xiuzhen Cheng. A measurement study of Model Context Protocol ecosystem. arXiv preprint arXiv:2509.25292, September 2025.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, 2025. arXiv:2403.07974.

Suozhao Ji, Baodong Wu, Zehao Wang, Lei Xia, Qingping Li, Ruisong Wang, Wenbo Ding, Zhenhua Zhu, Boxun Li, Guohao Dai, and Yu Wang. Infini Memory: Maintainable topic documents for long-term LLM agent memory. arXiv preprint arXiv:2606.10677, June 2026.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Jungo Kasai, Keisuke Sakaguchi, yoichi takahashi, Ronan Le Bras, Akari Asai, Xinyan Yu, Dragomir Radev, Noah Smith, Yejin Choi, and Kentaro Inui. Realtime qa: What's the answer right now? In *Advances in Neural Information Processing Systems*, volume 36, pp. 49025–49043. Curran Associates, Inc., 2023. doi: 10.52202/075280-2130. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/9941624ef7f867a502732b5154d30cb7-Paper-Datasets_and_Benchmarks.pdf.

Sachit Kuhar, Wasi Uddin Ahmad, Zijian Wang, Nihal Jain, Haifeng Qian, Baishakhi Ray, Murali Krishna Ramanathan, Xiaofei Ma, and Anoop Deoras. LibEvolutionEval: A benchmark and study for version-specific code generation. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 6826–6840, 2025.

Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration. *Empirical Software Engineering*, 23(1):384–417, 2018.

Chingkwun Lam, Jiaxin Li, Lingfei Zhang, and Kuo Zhao. Governing evolving memory in LLM agents: Risks, mechanisms, and the stability and safety governed memory (SSGM) framework. arXiv preprint arXiv:2603.11768, March 2026.

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. arXiv preprint arXiv:2507.03724, July 2025.

Zongxia Li, Zhongzhi Li, Yucheng Shi, Ruhan Wang, Junyao Yang, Zhichao Liu, Xiyang Wu, Anhao Li, Yue Yu, Ninghao Liu, Lichao Sun, Haotao Mi, and Leowei Liang. Long-Horizon-Terminal-Bench: Testing the limits of agents on long-horizon terminal tasks with dense reward-based grading. arXiv preprint arXiv:2607.08964, July 2026.

Junfeng Liao, Qizhou Wang, Jianing Zhu, Bo Du, Rui Yan, and Xiuying Chen. Belief memory: Agent memory under partial observability. arXiv preprint arXiv:2605.05583, May 2026.

Zeyu Leo Liu, Shrey Pandit, Xi Ye, Eunsol Choi, and Greg Durrett. CodeUpdateArena: Benchmarking knowledge editing on API updates. arXiv preprint arXiv:2407.06249, 2024.

Jinghao Luo, Yuchen Tian, Chuxue Cao, Ziyang Luo, Hongzhan Lin, Kaixin Li, Chuyi Kong, Ruichao Yang, and Jing Ma. From storage to experience: A survey on the evolution of LLM agent memory mechanisms. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 41622–41652, 2026.

Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, Baolong Bi, Yujun Cai, Jiazhi Liu, Mingyu Li, Zhong-Zhi Li, Duzhen Zhang, Chenlin Zhou, Jiayi Mao, Tianze Xia, Jiafeng Guo, and Shenghua Liu. A survey of context engineering for large language models. arXiv preprint arXiv:2507.13334, July 2025.

Diganta Misra, Nizar Islah, Victor May, Brice Rauby, Zihan Wang, Justine Gehring, Antonio Orvieto, Muawiz Sajjad Chaudhary, Eilif B. Muller, Irina Rish, Samira Ebrahimi Kahou, and Massimo Caccia. GitChameleon 2.0: Evaluating AI code generation against Python library version incompatibilities. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 46792–46831, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.2170. URL https://aclanthology.org/2026.acl-long.2170/.

Subhrangshu Nandi, Arghya Datta, Rohith Nama, Udita Patel, Nikhil Vichare, Indranil Bhattacharya, Shivam Asija, Arushi Gupta, Giuseppe Carenini, Jing Xu, Shayan Ray, Huzefa Raja, Aaron Chan, Francesco Carbone, Esther Xu Fei, Gaoyuan Du, Zuhaib Akhtar, Prince Grover, Sreyoshi Bhaduri, Weian Chen, Wei Zhang, Ming Xiong, Harshita Asnani, and Jeetu Mirchandani. SOP-Bench: Complex industrial SOPs for evaluating LLM agents. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, pp. 9604–9615, Jeju Island Republic of Korea, August 2026. ACM. doi: 10.1145/3770855.3817538.

Xuying Ning, Katherine Tieu, Dongqi Fu, Tianxin Wei, Zihao Li, Yuanchen Bei, Jiaru Zou, Mengting Ai, Zhining Liu, Ting-Wei Li, Lingjie Chen, Yanjun Zhao, Ke Yang, Bingxuan Li, Cheng Qian, Gaotang Li, Xiao Lin, Zhichen Zeng, Ruizhong Qiu, Sirui Chen, Yifan Sun, Xiyuan Yang,

Ruida Wang, Rui Pan, Chenyuan Yang, Dylan Zhang, Liri Fang, Zikun Cui, Yang Cao, Pan Chen, Dorothy Sun, Ren Chen, Mahesh Srinivasan, Nipun Mathur, Yinglong Xia, Hong Li, Hong Yan, Pan Lu, Lingming Zhang, Tong Zhang, Hanghang Tong, and Jingrui He. Code as agent harness. arXiv preprint arXiv:2605.18747, May 2026.

Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen Vinju. Breaking bad? Semantic versioning and impact of breaking changes in Maven Central: An external and differentiated replication study. *Empirical Software Engineering*, 27(3):61, 2022.

Siru Ouyang, Jun Yan, Yanfei Chen, Rujun Han, Zifeng Wang, Bhavana Dalvi Mishra, Rui Meng, Chun-Liang Li, Yizhu Jiao, Kaiwen Zha, Maohao Shen, Vishy Tirumalashetty, George Lee, Jiawei Han, Tomas Pfister, and Chen-Yu Lee. SkillOS: Learning skill curation for self-evolving agents. arXiv preprint arXiv:2605.06614, May 2026a.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, volume 2026, pp. 94327–94354, 2026b. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/980ea04d23d1f6908964eba2a74afe45-Paper-Conference.pdf.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. arXiv preprint arXiv:2310.08560, 2023.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pp. 1–22, San Francisco CA USA, October 2023. ACM. doi: 10.1145/3586183.3606763.

Vedant Patel. Supersede: Diagnosing and training the memory-update gap in LLM agents. arXiv preprint arXiv:2606.27472, June 2026.

Junjia Qi, Zichuan Fu, Jingtong Gao, Wenlin Zhang, Hanyu Yan, Xian Wu, and Xiangyu Zhao. LLM-as-Code: Agentic programming for agent harness. arXiv preprint arXiv:2606.15874, June 2026.

Lawrence R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

Preston Rasmussen, Pavlo Paliychuk, Travis Beauvais, Jack Ryan, and Daniel Chalef. Zep: A temporal knowledge graph architecture for agent memory. arXiv preprint arXiv:2501.13956, January 2025.

Romain Robbes, Mircea Lungu, and David Röthlisberger. How do developers react to API deprecation? The case of a Smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pp. 1–11, Cary North Carolina, November 2012. ACM. doi: 10.1145/2393596.2393662.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652, 2023.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7080–7106, Rabat, Morocco, March 2026. Association for Computational Linguistics. ISBN 979-8-89176-380-7. doi: 10.18653/v1/2026.eacl-long.333. URL https://aclanthology.org/2026.eacl-long.333/.

Huy Q. Tran, Dang H. Vu, Tuyen N. Dinh, Anh H. D. Nguyen, Anh N. H. Vu, Anh M. T. Bui, and Phuong T. Nguyen. Towards knowledge alignment in code LLMs: Contrastive unlearning for evolving APIs. arXiv preprint arXiv:2606.30810, June 2026.

Chong Wang, Kaifeng Huang, Jian Zhang, Yebo Feng, Lyuye Zhang, Yang Liu, and Xin Peng. LLMs meet library evolution: Evaluating deprecated API usage in LLM-based code completion. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 885–897. IEEE, 2025a.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024a.

Jiawei Wang, Li Li, Kui Liu, and Haipeng Cai. Exploring how deprecated Python library APIs are (not) handled. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 233–244, Virtual Event USA, November 2020. ACM. doi: 10.1145/3368089.3409735.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, December 2024b. doi: 10.1007/s11704-024-40231-1.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025b.

Tianxin Wei, Zhan Shi, Minhua Lin, Bing He, Zewen Liu, Yisi Sang, Yuanchen Bei, Xuying Ning, Jiaru Zou, Ting-Wei Li, Xiao Lin, Yanjun Zhao, Chi Wang, Benoit Dumoulin, Dakuo Wang, Jingrui He, and Hanqing Lu. Evo-Harness: Context-to-harness skill compilation for self-evolving agents. arXiv preprint arXiv:2608.15071, August 2026.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. LongMemEval: Benchmarking chat assistants on long-term interactive memory. In *International Conference on Learning Representations*, 2025.

Tongtong Wu, Weigang Wu, Xingyu Wang, Kang Xu, Suyu Ma, Bo Jiang, Ping Yang, Zhenchang Xing, Yuan-Fang Li, and Gholamreza Haffari. VersiCode: Towards version-controllable code generation. arXiv preprint arXiv:2406.07411, October 2024.

Xiaojun Wu, Cehao Yang, Honghao Liu, Xueyuan Lin, Wenjie Zhang, Zhichao Shi, Xuhui Jiang, Chengjin Xu, Jia Li, and Jian Guo. Bayesian-Agent: Posterior-guided skill evolution across LLM agent harnesses. arXiv preprint arXiv:2606.08348, June 2026.

Boming Xia, Liming Zhu, Zhenchang Xing, Qinghua Lu, Dino Sejdinovic, and Xiwei Xu. Harnessing agent skills: Architectural patterns and a reference architecture for skill-mediated LLM agents. arXiv preprint arXiv:2606.20631, May 2026.

Cheng Xu, Shuhao Guan, Derek Greene, and M.-Tahar Kechadi. Benchmark data contamination of large language models: A survey. arXiv preprint arXiv:2406.04244, June 2024a.

Jundong Xu, Qingchuan Li, Jiaying Wu, Yihuai Lan, Shuyue Stella Li, Huichi Zhou, Bowen Jiang, Lei Wang, Jun Wang, Anh Tuan Luu, Caiming Xiong, Hae Won Park, Bryan Hooi, and Zhiyuan Hu. EvoArena: Tracking memory evolution for robust LLM agents in dynamic environments. arXiv preprint arXiv:2606.13681, June 2026.

Rongwu Xu, Zehan Qi, Zhijiang Guo, Cunxiang Wang, Hongru Wang, Yue Zhang, and Wei Xu. Knowledge conflicts for LLMs: A survey. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 8541–8565, 2024b.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-Mem: Agentic memory for LLM agents. In *Advances in Neural Information Processing Systems*, volume 38, pp. 17577–17604, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang,

Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. arXiv preprint arXiv:2505.09388, May 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *International Conference on Learning Representations*, 2025.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/8a94ff6f922d995d7d3f4ebf4143e442-Paper-Conference.pdf.

Zeyu Zhang, Quanyu Dai, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model-based agents. *ACM Transactions on Information Systems*, 43(6):1–47, November 2025. doi: 10.1145/3748302.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024. doi: 10.1609/aaai.v38i17.29936.

Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and Yu Su. SkillWeaver: Web agents can self-improve by discovering and honing skills. arXiv preprint arXiv:2504.07079, April 2025a.

Junhao Zheng, Xidi Cai, Qiuke Li, Duzhen Zhang, ZhongZhi Li, Yingying Zhang, Le Song, and Qianli Ma. LifelongAgentBench: Evaluating LLM agents as lifelong learners. arXiv preprint arXiv:2505.11942, May 2025b.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. MemoryBank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19724–19731, 2024.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, Binyuan Hui, Niklas Muennighoff, David Lo, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations*, volume 2025, pp. 66602–66656, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/a6a90bcc2aa470c3871b2d39a67d26e8-Paper-Conference.pdf.

A Appendix

A.1 Details of the Bayesian model

This section collects the derivations and implementation choices behind §3.

Learning λ. We use continuous-time regime dynamics to advance beliefs over gaps of Δt periods (Eq. 5). We fix μ and update λ with a Gamma–Poisson model: each inferred entry into CHANGING adds one event, and stable exposure adds $\Delta t(1 - b_g)$. The posterior mean includes the prior event and exposure counts. The constants ε and ρ define one-period break probabilities, not rates. Over longer gaps, we repeat the validity update $s_k^-(z_g) = s_k + (1 - s_k)h_g$ once per period while advancing the regime belief.

Exact four-state update. Since both v_k and z_g are binary, an observation updates their four joint configurations exactly,

$$\Pr(v_k, z_g \mid y) \propto \Pr(v_k, z_g) \Pr(y \mid v_k), \tag{14}$$

and the beliefs used in the main text are its marginals,

$$s_k^+ = \Pr(v_k = 0 \mid y), \qquad b_g^+ = \Pr(z_g = \text{CHANGING} \mid y). \tag{15}$$

For the item this is the two-hypothesis update

$$s_k^+ = \frac{\bar{s}_k^- L(y \mid v_k = 0)}{\bar{s}_k^- L(y \mid v_k = 0) + (1 - \bar{s}_k^-) L(y \mid v_k = 1)}, \tag{16}$$

whose odds form is Eq. 9. The regime posterior of Eq. 10 must retain the two regime-specific validity probabilities $s_k^-(\text{STABLE})$ and $s_k^-(\text{CHANGING})$, which is why the prediction step carries them separately before marginalising.

Attribution. We attribute an outcome to an item when its call or classified error identifies it. Otherwise, each of the n items in context receives weight $1/n$, assuming equal responsibility.

Approximations. The four-state update is exact for one item and its regime. Sequential item updates approximate the joint subsystem posterior, and the expected-count update approximates the quality posterior by a Beta distribution.

Role of quality in the policy. Quality controls how strongly a failure raises the staleness belief through $(1 - q)/(1 - \bar{p}_k)$. The weight w_k reduces the contribution of stale-item failures to quality counts. Maintenance is triggered by estimated staleness and verification, rather than low quality alone.

A.2 HYPERPARAMETERS AND PRIORS

Table 5 lists the code defaults used in the main experiments.

Table 5: Priors and fixed constants of the model (§3).

Symbol	Meaning	Value	Set by
q	$\Pr(y{=}1 \mid v_k{=}0)$	0.10	fixed
α_0, β_0	prior on m_g, the mean quality of a subsystem	5.0, 1.0	items are born from verified examples
κ	pseudo-observations an item borrows from its group	5	fixed
ε	item break probability per period, regime stable	0.012	background drift of a release
ρ	extra break probability per period, regime changing	0.5	share of a subsystem a change touches
μ	CHANGING→STABLE rate	1 per period	heavy change lasts about one release
λ prior	Gamma$(\nu_0, \nu_0/\lambda_0)$ on STABLE→CHANGING	$\nu_0{=}2,\ \lambda_0{=}10^{-3}$	weak; learned online
s_0	initial $\Pr(v_k{=}0)$	0	item just written or verified
b_0	initial $\Pr(z_g{=}\text{CHANGING})$	0	fixed

We estimated ε, ρ and μ from release histories of XGBoost, OpenCV, SciPy and vLLM, using the fraction of functions affected by ordinary and heavy releases and the duration of heavy change. These libraries are outside both evaluation corpora and the generator calibration set. The generator was calibrated separately on scikit-learn, Django and xarray, using silent-change frequency and the concentration of changes within a subpackage. xarray also appears in MASKEDMIG (one migration), but no constant of Bayesian-Harness is measured on it. Appendix A.8 tests departures from the model's rates and structure.

A.3 THE WORLD GENERATOR AND ITS CALIBRATION

We first checked library drift using BigCodeBench tasks (Zhuo et al., 2025) on post-cutoff releases and release histories of scikit-learn, Django and xarray. The histories provided two generator targets:

about 20% of changes alter behaviour without changing the name or signature, and 55–82% of changes in heavy months occur within one subpackage.

The subsystem layout and the business vocabulary of ACMESDK are seeded from SOP-Bench (Nandi et al., 2026), whose enterprise procedures name the kinds of internal service an agent is asked to drive (billing, ledger, catalog, shipping and the rest); we take that seed and have Claude Code (Fable 5.1) write the package, its 13 kernel templates and its release history generator, then calibrate the drift statistics on the real repository histories above. No SOP-Bench task, tool implementation or trajectory is reused.

ACMESDK is a fictional company-internal SDK with 12 subsystems (auth, billing, catalog, export, identity, ledger, metrics, notify, scheduler, search, shipping, storage) of 30–40 pure-Python functions each, 412–417 functions per seed. Every function is an instance of one of 13 kernel templates (linear scaling, percentage, clamping, tiering, id formatting, joining, top-k selection, aggregation, field picking, date shifting, checksums, bucketing and week alignment), parameterised by knobs (rounding, tier boundaries, units, separators) and a return shape (scalar, dict, tuple or list), with names drawn from verb and noun vocabularies per subsystem. The package ships as sourceless bytecode behind proxy modules: `dir()` is empty, signatures are `(*args, **kwargs)`, docstrings are absent, errors are generic, and the dispatch table lives in a compressed blob so that `strings` on the bytecode reveals no names. At monthly test time, the agent's memory items are the only description it has.

The SDK ships 36 monthly releases. Change types in the background trickle and in the vertical heavy months are drawn with weights rename 30, parameter change 25, return shape 15, removal with a named successor 10 and silent semantics 20. Three layers produce them: a background trickle at 1% of functions per month; six *vertical* heavy months, each rewriting 40–60% of one subsystem, at least three months apart; and two *horizontal* ones that cut across subsystems, one renaming the `token` parameter of every authenticated function to `credential`, one switching every money-tagged function from major to minor units. The two horizontal months are not drawn from these weights. A function changes at most three times; a name is never reused within a function's history. Each release publishes a changelog with one line per changed function and its current usage; this is the world-given external knowledge that a repair may read once its memory item has been selected. Irreversible knobs (modulo, right-alignment, de-duplication) are excluded from silent drift so that every change can, in principle, be compensated in code.

Each function has three task variants, each a legacy snippet with frozen assertions written against release 1.1.0. An in-process simulation of every release confirms which variants break in which month, and a spot check of 40 tasks in three months against the real sandbox found no disagreement. Monthly tests are drawn from a Zipf(1.0) popularity over functions, up to 10 per month, giving 330–346 monthly tests per seed. At month 1 the agent holds one verified usage memory item per function. The sandbox pins `PYTHONPATH` to the release directory and charges one unit per command.

We also ran the policies on MASKEDMIG, 157 real migrations: legacy solutions to tasks from 18 Python libraries (numpy 42, langchain-core 23, sympy 22, matplotlib 19, pandas 11, torch 11, networkx 5, redis 5, datasets 5, huggingface-hub 3, sphinx 2, pillow 2, black 2, and one each from pymongo, gradio, pytest, xarray and dash) whose API was removed by its maintainers and replaced by a named successor in a release between April 2025 and August 2026. Monthly tests are spread over a nine-quarter timeline, zero to three quarters after the release that breaks the memory item, so that different callers meet the same break at different times. The 157 monthly tests are the same in every seed; a seed draws, for each migration, the quarter within that window in which it is examined, so the three seeds differ in when each break is met, not in what is asked. Contamination is controlled by masking: from its release on, every successor is renamed to a hash, consistently in the installed package, the changelog the repair step may read and the monthly test, so the agent cannot recall the new name from pre-training and a monthly test passes only if the frozen assertions hold and the masked successor is used. The real packages retain introspection, so masking prevents recall of successor names but still allows discovery through sandbox exploration.

Each quarter the agent spends 24 units (probe 1, repair 3) and then sits the quarter's monthly tests; each library is one subsystem with its own regime z_g, and the beliefs advance by $\Delta t = 3$ monthly periods each quarter. One memory item per migration gives 157 memory items, each examined

once, and a library's legacy memory items break in the quarter of its release and never recover. §4.4 reports the outcome (Table 1).

A.4 PROMPTS

The monthly-test agent and the repair operator are the same model with the following system prompts; both are held fixed across all methods. The monthly-test prompt is followed by the injected memory items (the agent's own memory item for the function and its two most-used subsystem siblings), the failing snippet and the budget; the repair prompt is followed by the stale memory item, the subsystem's changelog entries since the memory item was last valid, and the budget. Those entries are assembled only after the maintenance policy has picked the memory item, so no policy can use them to pick. Note that in the prompt, when explaining its task to the agent, memory items are called "notes" or "your notes".

Monthly-test agent, system prompt

```
You maintain legacy code that uses acmesdk, an internal SDK of your company.  A small
snippet that worked on an older acmesdk release now FAILS on the current release.  Repair
it so the ORIGINAL assertions pass here.

acmesdk is internal:  it is not on the public internet and you have no prior knowledge of
it.  Its modules expose no listing (dir() is empty), no docstrings, and error messages are
generic.  Your notes are the only description you have.  Trust them, and verify with a
command when unsure.

Hard rules:

 - You must NOT modify, remove, or reorder any `assert` line:  they are the spec.  Change
   only the other lines (imports, setup, the call, any post-processing needed so the asserts
   still hold).  If a note tells you the function's output changed (new shape, new unit, new
   order), keep the asserts and compensate in code:  unwrap, rescale, reorder, or convert
   the result back to what the asserts expect.
 - The repaired snippet must still call acmesdk to do the work (the same function or its
   current replacement).  Recomputing the values without acmesdk counts as failure.
 - Work it out with bash (`python -c "..."` or write and run a file), exactly ONE bash code
   block with ONE command per response; each command costs one unit of a small budget.

When your repaired snippet passes here (or budget is gone), finish with:

FIXED {"code":  "<the full repaired snippet, original assert lines intact>"}
```

Repair operator, system prompt

```
You maintain a notebook about acmesdk, your company's internal SDK, for your future self.
One note's example snippet no longer runs on the current release.  You are given the
release notes for that subsystem since the note was last valid.

Repair the example the way you would repair legacy code.  Keep every `assert` line of the
stale example EXACTLY as it is, and change the other lines so that the asserts pass on the
current release:  call the function by its current name, pass parameters the way it now
expects, and if its output changed (new shape, new unit, new order) add the post-processing
that turns the new output back into the old value.  The repaired example is then a worked
recipe for every legacy caller of this function, which is what your future self needs most.
In the note text state what changed and the old-to-new mapping in one or two sentences.

Test with bash first:  exactly ONE bash code block with ONE command per response, each
command costs one unit of a small budget.

When the repaired example passes here with the original asserts, finish with:

REPAIRED {"snippet":  "<repaired example, original assert lines intact>", "text":  "<what
changed + old-to-new mapping>"}
```

A.5 FULL RESULTS ON ACMESDK

Tables 6, 7, 8 and 9 report results by model and budget, drift type, period, and budget and model size.

Results by drift type, model and time. Bayesian-Harness leads the maintenance baselines overall for all three models in Table 7. Renamed drift shows the clearest gains: success is 57.3% for Qwen3.5-27B, 33.9% for Flash-Lite and 64.2% for Gemini 3.1 Pro. For Qwen3.5-27B, overall

Table 6: **Monthly test success (↑) on ACMESDK by model and monthly budget**, three seeds pooled, ± s.d. across seeds. Bold marks the best maintenance policy per column and block; Δ is the gain of Bayesian-Harness over *None*. None and Static spend no budget, so every block repeats the same runs of the two. Oracle is the staleness-informed diagnostic reference defined in §4.3.

Method	Qwen3.5-4B	Qwen3.5-9B	Qwen3.5-27B	Gemini 2.5 Flash-Lite	Gemini 3.1 Pro
Budget 12 units per month					
Oracle	28.9±5.1	44.8±4.8	67.4±5.6	34.8±0.7	84.0±1.2
None	15.8±3.6	21.1±5.9	41.5±3.7	13.4±1.5	54.9±4.5
Static	19.5±5.9	29.7±2.1	44.3±0.6	19.1±2.7	56.9±2.6
Random	24.4±3.1	35.9±1.8	54.1±4.3	26.2±2.1	66.4±2.4
Bayesian-Agent	26.4±4.0	36.8±3.6	56.1±6.5	26.6±2.5	69.3±2.4
TTL	26.6±3.6	39.8±4.6	62.6±3.4	30.8±4.4	74.3±3.3
Bayesian-Harness	**33.7**±4.0	**46.7**±5.5	**67.0**±3.4	**35.5**±5.2	**79.0**±2.0
Δ *vs. None*	+17.9 ↑	+25.6 ↑	+25.5 ↑	+22.1 ↑	+24.1 ↑
Budget 24 units per month					
Oracle	40.2±1.7	54.5±7.5	78.1±3.2	46.1±2.9	92.6±1.6
None	15.8±3.6	21.1±5.9	41.5±3.7	13.4±1.5	54.9±4.5
Static	19.5±5.9	29.7±2.1	44.3±0.6	19.1±2.7	56.9±2.6
Random	30.7±5.4	42.5±1.6	60.4±3.8	32.7±3.0	71.6±1.4
Bayesian-Agent	30.1±4.2	41.0±5.7	64.7±3.8	33.5±2.4	77.3±1.7
TTL	36.2±2.7	51.2±4.0	62.9±4.4	37.0±1.5	77.0±2.2
Bayesian-Harness	**40.7**±3.8	**52.8**±7.7	**74.6**±1.4	**41.3**±2.2	**83.7**±1.2
Δ *vs. None*	+24.9 ↑	+31.7 ↑	+33.1 ↑	+27.9 ↑	+28.8 ↑
Budget 48 units per month					
Oracle	49.7±4.4	63.7±9.8	90.2±1.0	55.6±3.5	97.9±1.3
None	15.8±3.6	21.1±5.9	41.5±3.7	13.4±1.5	54.9±4.5
Static	19.5±5.9	29.7±2.1	44.3±0.6	19.1±2.7	56.9±2.6
Random	40.6±2.9	50.8±4.9	70.2±3.1	40.3±0.4	79.1±0.8
Bayesian-Agent	37.7±2.9	50.2±6.7	71.4±3.2	39.8±1.4	83.0±1.0
TTL	39.6±2.1	52.3±3.0	66.3±4.7	38.3±2.2	78.1±1.8
Bayesian-Harness	**45.5**±1.2	**59.9**±4.7	**76.8**±1.7	**48.3**±1.3	**86.0**±1.1
Δ *vs. None*	+29.7 ↑	+38.8 ↑	+35.3 ↑	+34.9 ↑	+31.1 ↑

success rises from 70.9% in months 2–12 to 74.7% in months 25–36, while Oracle moves from 81.6% to 75.0%.

Per-seed results. Table 10 gives the per-seed numbers behind the pooled blocks of Table 7. Per seed, counting the renamed monthly tests that exactly one of the two policies passes, Bayesian-Harness beats Bayesian-Agent 33 to 1, 25 to 8 and 25 to 3 with Qwen3.5-27B.

Mechanical repairs. For explicit renames, removals with named successors and parameter renames, we also tested mechanical repairs at one unit each, keeping model-based repairs at three units. On seed 1, renamed/overall success was 63.8/75.4 for Bayesian-Harness, 49.7/68.2 for Bayesian-Agent, 53.0/68.8 for TTL and 81.9/85.5 for Oracle. Mechanical repairs succeeded on 85% of renames, 77% of removals and 72% of parameter renames, but no shape or semantic changes. The main experiments use a uniform three-unit repair cost.

A.6 Quality ablation

We freeze $\bar{p}_k$ to test quality estimation (Table 11). At $\bar{p}_k = q$, pass/fail outcomes provide no validity evidence. At $\bar{p}_k = 1$, every failure implies staleness. The intermediate setting $\bar{p}_k = 0.5$ uses a fixed reliability estimate. We use a monthly budget of 12, where unnecessary probes are more costly; the differences narrow at budget 24. Estimated quality gives the highest renamed and overall success among these variants.

Table 7: **Monthly test success (↑) at budget 24 by drift type.** ± is the s.d. across three seeds; the column counts cover all three blocks. Bold marks the best policy per column and block, Oracle excluded.

Method	renamed (n=363)	param (n=401)	behav. (n=255)	all (n=1019)
Qwen3.5-27B				
Oracle	63.1±9.2	78.6±2.0	98.8±1.3	78.1±3.2
TTL	40.2±7.3	61.3±8.6	97.6±1.3	62.9±4.4
Bayesian-Agent	37.7±4.8	68.8±6.3	96.5±1.6	64.7±3.8
Random	28.4±5.1	65.8±7.9	97.3±0.3	60.4±3.8
Static	0.0±0.0	49.9±11.5	**98.4**±3.0	44.3±0.6
None	0.0±0.0	43.1±7.1	98.0±0.6	41.5±3.7
Bayesian-Harness	**57.3**±5.7	**75.3**±0.3	98.0±1.9	**74.6**±1.4
Gemini 2.5 Flash-Lite				
Oracle	35.0±5.4	56.4±4.6	45.9±0.8	46.1±2.9
TTL	27.5±3.9	46.4±2.4	35.7±7.6	37.0±1.5
Bayesian-Agent	18.5±4.9	44.1±2.4	38.0±7.8	33.5±2.4
Random	18.7±4.4	44.4±5.9	34.1±4.5	32.7±3.0
Static	0.0±0.0	27.9±6.3	32.5±8.3	19.1±2.7
None	0.0±0.0	13.7±2.9	32.2±7.6	13.4±1.5
Bayesian-Harness	**33.9**±4.3	**48.4**±2.4	**40.8**±3.9	**41.3**±2.2
Gemini 3.1 Pro				
Oracle	84.8±6.1	95.5±0.3	99.2±0.7	92.6±1.6
TTL	50.4±7.4	86.5±5.6	**100.0**±0.0	77.0±2.2
Bayesian-Agent	46.3±2.1	**91.5**±1.8	99.2±0.7	77.3±1.7
Random	31.7±6.2	90.0±3.3	99.6±0.7	71.6±1.4
Static	0.3±0.5	80.8±8.9	**100.0**±0.0	56.9±2.6
None	0.3±0.5	75.6±3.1	**100.0**±0.0	54.9±4.5
Bayesian-Harness	**64.2**±6.2	91.0±1.9	**100.0**±0.0	**83.7**±1.2

Table 8: **Monthly test success (↑) by test month, Qwen3.5-27B, budget 24.** The 1,019 tests of Table 7 split by time, ± s.d. across three seeds; early months have fewer tests. Bold marks the best policy per column.

Method	M02–12 (n=299)	M13–24 (n=360)	M25–36 (n=360)
Oracle	81.6±9.2	78.3±6.0	75.0±2.9
TTL	60.2±7.8	63.3±8.7	64.7±2.7
Bayesian-Agent	67.9±4.5	65.3±9.4	61.4±4.6
Random	62.2±5.2	61.7±8.7	57.5±3.6
Static	46.2±9.1	42.8±3.9	44.2±7.5
None	40.1±5.2	43.3±6.5	40.8±8.2
Bayesian-Harness	**70.9**±3.8	**77.5**±3.6	**74.7**±1.9

Table 9: **Monthly budget (Qwen3.5-27B, top) and model (budget 24, bottom).** Cells are success on renamed drift / all monthly tests (↑), three seeds pooled, ± s.d. across seeds; the Qwen3.5-27B entry of the bottom block is the 24-unit row of the top block and is not repeated. Bold marks the best policy per column and block, Oracle excluded.

	Oracle	Bayesian-Harness	TTL	Bayesian-Agent	Random
Monthly budget, Qwen3.5-27B					
12 units	43.3±8.5 / 67.4±5.6	**46.0**±6.5 / **67.0**±3.4	38.8±8.3 / 62.6±3.4	21.8±8.5 / 56.1±6.5	18.5±1.7 / 54.1±4.3
24 units	63.1±9.2 / 78.1±3.2	**57.3**±5.7 / **74.6**±1.4	40.2±7.3 / 62.9±4.4	37.7±4.8 / 64.7±3.8	28.4±5.1 / 60.4±3.8
48 units	84.3±4.2 / 90.2±1.0	**60.3**±6.6 / **76.8**±1.7	45.7±10.4 / 66.3±4.7	50.4±10.1 / 71.4±3.2	44.9±7.3 / 70.2±3.1
Model size, budget 24					
Qwen3.5-9B	41.0±10.9 / 54.5±7.5	**44.1**±11.1 / **52.8**±7.7	40.8±10.0 / 51.2±4.0	18.5±6.3 / 41.0±5.7	19.6±3.7 / 42.5±1.6
Qwen3.5-4B	31.1±3.0 / 40.2±1.7	**36.4**±3.3 / **40.7**±3.8	29.2±4.2 / 36.2±2.7	15.4±2.7 / 30.1±4.2	17.6±3.4 / 30.7±5.4

Table 10: Per-seed results at budget 24. Cells are success on renamed drift / overall.

Agent	Seed	Oracle	Bayesian-Harness	TTL	Bayesian-Agent	Static	None
Qwen3.5-27B	1	72.5 / 81.2	63.1 / 75.7	47.7 / 67.3	41.6 / 65.0	0.0 / 44.2	0.0 / 39.6
	2	57.4 / 78.1	53.7 / 74.9	34.3 / 62.7	38.0 / 68.2	0.0 / 44.9	0.0 / 45.8
	3	55.7 / 74.8	52.8 / 73.0	35.8 / 58.5	32.1 / 60.6	0.0 / 43.6	0.0 / 39.1
Flash-Lite	1	32.9 / 45.4	36.9 / 43.6	31.5 / 38.7	22.8 / 36.1	0.0 / 21.1	0.0 / 15.0
	2	31.5 / 43.7	28.7 / 39.4	25.0 / 36.2	13.0 / 32.7	0.0 / 16.0	0.0 / 12.0
	3	41.5 / 49.4	34.9 / 40.9	24.5 / 36.1	17.9 / 31.5	0.0 / 20.3	0.0 / 13.3
Gemini 3.1 Pro	1	89.9 / 94.2	70.5 / 85.0	57.7 / 79.5	48.3 / 75.4	0.0 / 54.0	0.0 / 49.7
	2	77.8 / 91.0	60.2 / 83.4	43.5 / 75.5	45.4 / 78.4	0.0 / 58.9	0.0 / 57.7
	3	84.9 / 92.7	59.4 / 82.7	47.2 / 76.1	44.3 / 78.2	0.9 / 57.9	0.9 / 57.3

Table 11: **The quality variable.** Bayesian-Harness with $\bar{p}_k$ frozen instead of estimated; Qwen3.5-27B, three seeds, budget 12. Cells are success on renamed drift (n=363) / all monthly tests (n=1,019) (↑), ± s.d. across seeds.

	renamed / all
Bayesian-Harness (quality estimated)	**46.0**±6.5 / **67.0**±3.4
$\bar{p}_k$ fixed at 0.5	43.0±4.9 / 65.5±0.1
$\bar{p}_k$ fixed at $q = 0.1$ (test outcome uninformative)	39.7±6.7 / 63.3±1.6
$\bar{p}_k$ fixed at 1 (no quality variable)	36.9±11.0 / 63.8±3.6

A.7 Component ablation on MASKEDMIG

Table 2 shows that on ACMESDK every component of Bayesian-Harness earns its place. The same conclusion holds on MASKEDMIG. Table 12 switches off one component at a time, with each migration examined in three successive quarters instead of one. The longer exposure lifts every variant, which is why the full model reads 56.3 here and 48.4 in Table 1. This three-quarter protocol differs from the single-test protocol of that table, so we report within-protocol comparisons.

Removing any component lowers test success by 4.9–6.7 points. The shared regime has the largest effect (6.7 points), followed by quality pooling (6.0), recovery (5.2), and the error channel (4.9).

The result also shows that the components are useful beyond the injected bursts of ACMESDK. Even though real releases in MASKEDMIG are more regular and each library carries fewer memory items than a subsystem in ACMESDK, sharing evidence through the shared regime and the quality pool remains important. Recovery and failed-call evidence provide comparably sized gains, indicating that both temporal dynamics and direct error observations matter over the three-quarter evaluation window.

Table 12: **Ablation of Bayesian-Harness on MASKEDMIG**, Qwen3.5-27B, budget 24 per quarter, three seeds, one component off per row. Monthly test success (↑) over the 157 migrations with each examined in three successive quarters, ± s.d. across seeds. All four components improve aggregate success, consistent with Table 2 on ACMESDK.

	Monthly test success	Δ vs full
Bayesian-Harness	**56.3**±1.9	
no recovery	51.1±0.7	−5.2
no error channel	51.4±1.9	−4.9
no pooling	50.3±3.3	−6.0
no shared regime	49.6±1.9	−6.7

A.8 Model mismatch

Table 4 in §4 tests incorrect groups, different drift rates, cross-subsystem changes, scheduled bursts and longer changing periods. Bayesian-Harness leads the tested baselines in each setting.

Misfiling 30% or all memory items tests whether the policy depends on correct subsystem assignments. Rate tests use either no heavy months with 2.4% background drift or 11 heavy months with 0.5% background drift. The hidden-axis test uses eight events that change a shared kernel template across subsystems, plus two subsystem-specific heavy months. In seed 1 its 422 drift events come

from the kernel changes (196), the two subsystem-specific heavy months (35), two package-wide heavy months (105) and background drift (86). Timing tests place heavy months five months apart or extend changing periods to six and three months. Every row of the table pools three seeds.

A.9 BUDGET EFFICIENCY

With 24 units a month, Bayesian-Harness matches or exceeds the strongest maintenance baseline at 48 units on all five models in Table 6: 40.7 versus 40.6 on Qwen3.5-4B, 52.8 versus 52.3 on Qwen3.5-9B, 74.6 versus 71.4 on Qwen3.5-27B, 41.3 versus 40.3 on Flash-Lite, and 83.7 versus 83.0 on Gemini 3.1 Pro.

A.10 PROTOCOL OF THE SEPARABILITY TEST

This section gives the protocol behind Table 3 (§4.6).

In the experiments above, we test staleness but do not include memory items that are low-quality while still valid. To test whether we can separate the two, we assign a random half of the functions in ACMESDK to *cheap* repairs. We patch and verify their snippets, but the rewritten memory item records only the function's current name, with no usage example and no explanation. For the other half, we use full model repairs. We otherwise follow the main protocol. We introduce low-quality memory items through repairs because we draw monthly tests from broken snippets: we never examine an initially low-quality memory item while it is valid, and we rewrite it once its interface breaks.

We label a memory item as *stale* if its interface changed after its last repair, and as *low-quality* if its last repair was cheap. We use the release history and repair log for these labels. Before comparing policies, we check that the agent passes fewer tests with cheaply repaired memory items than with fully repaired memory items while both are valid (Table 3, caption).

We measure three things. First, we compare each policy's mean ranking score on low-quality but valid memory items and good but stale memory items. We also compare its quality estimates on cheaply and fully repaired valid memory items. Second, we measure how often the policy probes valid memory items and how many stale memory items it verifies within one period of breaking. Third, we report monthly test success on renamed drift and on all tests. In §3.5, we use quality only to update beliefs about staleness, not to trigger rewrites. By accounting for failures due to low quality, we aim to avoid unnecessary probes and false alarms about changes in the subsystem.

We compare Bayesian-Harness with three alternatives. *Bayesian-Agent* uses a single score. *Two counters* uses a TTL clock and a running success rate; it treats a failure as evidence of staleness only when the memory item's success rate exceeds its subsystem mean. In the third alternative, we freeze quality at $\bar{p}_k = 1$. Through these comparisons, we test whether we need separate quality and staleness estimates, and whether the probabilistic model helps beyond keeping two counters.

We run the ACMESDK comparison with Qwen3.5-27B at monthly budgets of 12 and 24, and with Qwen3.5-9B at budget 12. We also apply cheap repairs to half of the migrations in MASKEDMIG: the snippet is moved to the masked successor and verified, but the memory item's text records the public successor name rather than the deployment's masked symbol. We use three test variants per migration, for 471 tests per seed, so we can examine valid memory items repeatedly and accumulate quality counts $(n_k^{\mathrm{s}}, n_k^{\mathrm{f}})$. Each memory item breaks only once in this corpus, so we do not observe low-quality memory items that later become stale.

Finally, we test whether we should also use quality to choose rewrites. Each period we reserve a third of the budget for full rewrites of memory items the policy believes valid and has repaired before, taking those whose $\bar{p}_k$ is below their subsystem mean in order of $(1 - \bar{p}_k)$ weighted by usage; when no memory item qualifies the reserve returns to probing, and any budget left after probing also goes to rewrites. We compare this variant with the main policy to see whether rewriting low-quality memory items improves monthly test success beyond ranking by staleness.

A.11 A THIRD KNOWLEDGE TYPE: COMMAND-LINE TOOL MEMORY ITEMS

The two corpora above hold memory items about a Python API. To test the policies on a different kind of knowledge we give the ACMESDK worlds a command-line surface, ACMECTL. The same functions, releases, drift events and changelogs are exposed as `acmectl <subsystem> <command>` with JSON arguments and JSON output; a rename becomes a renamed command, a parameter rename a renamed flag, a return-shape change a changed JSON output, and a removal a removed command with a named successor. A memory item is a shell script with a `test` line comparing the command's output with the expected value, and a monthly test is a broken shell script that must pass its original `test` lines while still calling ACMECTL; probes and grading run the scripts with `bash -e`. The month-by-month break pattern of every task is identical to the Python surface, so the two surfaces sit the same tests in the same months. Table 13 reports Qwen3.5-27B at budget 24 over the three seeds.

Table 13: **Monthly test success (↑) on the ACMECTL command-line surface of ACMESDK**, Qwen3.5-27B, budget 24, three seeds pooled (n=363 renamed, 1,019 all), ± s.d. across seeds. Bold marks the best maintenance policy.

Method	renamed	all
Oracle	55.4±3.3	69.2±1.5
None	0.0±0.0	27.7±2.3
Static	0.0±0.0	32.7±2.8
Random	24.5±3.2	51.6±4.4
Bayesian-Agent	33.9±1.7	55.3±3.8
TTL	39.9±1.1	57.5±4.2
Bayesian-Harness	**53.4**±0.5	**65.9**±2.0