SKILL.NB: SELECTIVE FORMALIZATION AND GATED EXECUTION FOR DURABLE AGENT WORKFLOWS

Anonymous authors
Paper under double-blind review

ABSTRACT

AI agents increasingly convert past experience into reusable artifacts such as code, workflows, and procedural memories. Reuse improves efficiency but creates a lifecycle reliability problem: artifacts that succeed once may fail under environment drift, underspecified tasks, or changing task distributions, especially in web automation. We introduce SKILL.nb, a framework for governing reusable agent workflows through evidence-calibrated lifecycle policies. Its key mechanism is *selective formalization*: execution evidence decides which workflow steps should become executable code, which should remain natural-language-guided, and when those choices should be revised. SKILL.nb stores workflows as auditable, versioned notebooks that interleave natural-language guidance, multi-language executable cells, validation gates, fallback paths, and multimodal evidence such as outputs, screenshots, and error traces. At runtime, SKILL.nb performs *gate-conditioned execution*: unlike all-or-nothing scripts, each step can execute code when its gates validate, or fall back locally to an NL procedure or step intent when drift invalidates the executable realization. Cell-level records of attempted realizations, gate outcomes, outputs, screenshots, and fallbacks make both workflow updates and executions auditable. On WebArena-Verified, SKILL.nb achieves 53.7% success in an online round, 3.9 percentage points above the strongest baseline. Across three re-executions, it retains 91.7% of initially successful tasks, 15.5 points above the next best method. Under bounded repair, it recovers 72.9% of subsequent failures while limiting post-repair regressions to 4.2%, compared with 15.0–17.0% regression rates for persistent baselines. It also leads the compared methods on Mind2Web cross-website and cross-domain splits. In a realistic GitLab migration test, SKILL.nb preserves performance when reusing frozen state learned on GitLab 15.7, with frozen-versus-fresh target-version gaps of only -1.7 points on GitLab 16.11 and $+0.6$ points on GitLab 18.9; the least-degraded persistent baseline drops by 10.6–11.1 points. These results identify lifecycle governance and gate-conditioned execution as reliability axes beyond one-shot task success. Code, data, and evaluation scripts will be released after blind review.

1 INTRODUCTION

AI agents increasingly produce and rely on durable external artifacts such as code, workflows, and procedural memories (Liu et al., 2024; Wu et al., 2024; Wang et al., 2025b; Ouyang et al., 2025). As these artifacts are reused, the central question shifts from one-shot task completion to whether the artifact remains effective as conditions change, such as when the target site for a web agent changes its UI or data layout (Levy et al., 2025). Recent memory and workflow systems treat agent experience as reusable (Zheng et al., 2024; Zhao et al., 2024; Fang et al., 2025; Wang et al., 2025b; Ouyang et al., 2025), but repeated use introduces software-maintenance lifecycle concerns (Rahman et al., 2023): artifacts need versioning, validation, repair, regression control, and retirement when assumptions fail. Existing memory-centric systems often reuse experience as prompt context, while workflow and artifact systems provide execution, validation, versioning, or state-management primitives (Wang et al., 2025b; Ouyang et al., 2025; Liu et al., 2025a; Geng & Chang, 2025; Li et al., 2025; Mohammadi et al., 2026). These mechanisms are typically studied separately rather than as a joint lifecycle policy. For sustained automation, the question is not merely how to recall prior experience, but how accumulated execution evidence should govern reusable

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.

workflows: when to promote a candidate, when to formalize a step, when to repair or demote brittle realizations, and when to retire a workflow whose assumptions no longer hold.

We study this lifecycle problem in web environments because they provide a high-drift setting for evaluating learned artifacts. Site changes are outside the automation system's control: unlike desktop applications that can often be version-pinned, or APIs that typically expose explicit contracts and announce breaking changes, web interfaces can drift without preserving automation anchors such as selectors or layouts. While APIs are often the preferred automation interface, many real workflows still depend on UI-level interaction (Zhang & Zhang, 2024). Automating web workflows therefore forces a formalization choice: which steps should be hardened into code, and which should remain natural-language-guided (NL-guided) when interface drift would make repeated repair too costly. Leaving steps NL-guided preserves flexibility but weakens artifact-level control, while hardening them into code makes reuse more controlled but can become brittle under drift. Their validity therefore depends both on a task-level procedure and on an environment-dependent realization whose assumptions can break. For durable workflows, formalization is therefore lifecycle-governed. The system must decide when to create or retire a workflow, which steps remain NL procedures versus executable code, and when execution evidence warrants revising those choices as interfaces drift.

We introduce SKILL.nb, a framework for learning evidence-based lifecycle policies over reusable agent workflows. Its central mechanism is *selective formalization*: using execution evidence to decide which steps remain NL-guided, which become executable code, and when those choices should be revised. SKILL.nb couples this with *gate-conditioned execution*: runtime gates decide whether to execute code or fall back to an NL procedure. Each workflow is represented as an auditable, versioned notebook that interleaves NL guidance, executable code, validation gates, fallback paths, and cell-level evidence. Appendix A.1 visualizes how a provisional task notebook is promoted into a released workflow. Offline lifecycle learning aggregates traces into evidence counts for workflow creation and step formalization, and into accepted-repair signals, including repair counts and token-weighted repair burden as a proxy for instability in step demotion and workflow retirement. Thresholds are calibrated by replaying candidate lifecycle policies on historical workflows, selecting low-maintenance policies whose validation-failure rate stays within budget. This calibration uses accumulated execution evidence rather than hidden evaluator labels, and remains replay-relative rather than a guarantee under future interface shifts. Because web drift makes hardened code useful but brittle, the representation must preserve execution evidence, not just instructions. Otherwise, maintenance cannot tell whether code was merely suggested or actually ran, which gates validated it, what evidence it produced, or when execution fell back. This raises a natural question: why not use a conventional `SKILL.md` file with instructions and code snippets? Markdown can describe code, but it does not by itself record execution boundaries, gate outcomes, fallbacks, or cell-local evidence. SKILL.nb makes code execution first-class and auditable by storing these objects as versioned notebooks. Appendix A.3 compares the two representations along four axes: executable workflow state, validation feedback, evidence retention, and failure localization.

Empirically, SKILL.nb improves online task performance and controlled repeated-use reliability. On WebArena-Verified (El hattami et al., 2025), it achieves the highest single-round success rate among compared methods—CodeAct (Wang et al., 2024b), $\text{AWM}_{\text{online}}$ (Wang et al., 2025b), and ReasoningBank (Ouyang et al., 2025)—at 53.7%, outperforming the next-best method by 3.9 percentage points (paired McNemar, $p = 0.029$). In controlled repeated execution, SKILL.nb retains 91.7% of initially successful WebArena-Verified tasks across three re-executions, 15.5 percentage points above the next best method. Under bounded repair, it recovers 72.9% of subsequent failures while limiting post-repair regressions to 4.2%, compared with 15.0–17.0% for persistent baselines under their native update paths. As a secondary transfer evaluation on Mind2Web (Deng et al., 2023), SKILL.nb leads the compared methods on all four metrics in both the cross-website and cross-domain splits, with step success reaching 38.1% and 39.7%, respectively. In a controlled GitLab migration test, SKILL.nb preserves target-version performance when reusing frozen state learned on GitLab 15.7, while persistent-memory baselines degrade by 10.6–14.4 points. These results show that governed workflow artifacts improve task completion, reuse, repair, and regression control under those evaluated protocols.

Our key contributions are summarized as follows:

- We formulate durable web-agent automation as *selective formalization* and lifecycle governance for reusable, executable artifacts that must be formalized, validated, repaired, and retired as assumptions fail.
- We present SKILL.nb, a notebook-native framework that couples selective formalization with gate-conditioned execution, enabling versioned workflows that mix NL guidance, executable cells, validation gates, fallback, and auditable multimodal execution evidence.
- We evaluate SKILL.nb on WebArena-Verified and Mind2Web, showing the highest single-round performance among compared methods and improved retention, repair recovery, and regression control in repeated WebArena-Verified execution, and include a controlled GitLab version-drift protocol comparing fresh runs with old-state reuse after application migration.

2 Related Work

Agent Experience Memory. Memory is an essential module for agents (Zhang et al., 2024), with representations ranging from virtual paging (Packer et al., 2023) and structured graphs (Chhikara et al., 2025; Xu et al., 2025) to hierarchical working memory and consolidation mechanisms (Hu et al., 2025; Tan et al., 2025; Zhong et al., 2024). A complementary line of work focuses on reusing past experience for future tasks. For instance, Synapse retrieves raw trajectories as in-context exemplars (Zheng et al., 2024), while ExPeL and MemP distill experience into procedural insights and memory (Zhao et al., 2024; Fang et al., 2025). Extending this, Agent Workflow Memory (AWM) and ReasoningBank abstract execution traces into reusable workflows and reasoning strategies (Wang et al., 2025b; Ouyang et al., 2025). These systems motivate the use of accumulated experience. SKILL.nb represents that experience as versioned workflows with step-level intents, NL procedures, optional executable code, observable-state gates, and execution evidence.

Self-Evolving Agents. Continuous adaptation is a core desideratum for autonomous agents (Gao et al., 2026; Liang et al., 2024). To achieve cross-task evolution, systems maintain refined causal abstractions (CLIN (Majumder et al., 2023)), construct training curricula (Su et al., 2025), or consolidate transferable reasoning strategies (ICE (Qian et al., 2024), ChemAgent (Tang et al., 2025), Contextual Replay (Liu et al., 2025b)). Voyager builds an executable skill library and uses model-based task verification before inserting a skill (Wang et al., 2024a), while TroVE induces, reuses, and periodically trims a toolbox of program functions (Wang et al., 2024c). Agent Skill Induction (ASI) induces executable web skills and checks trajectories for model-judged task success, skill use, and environment changes before admission (Wang et al., 2025a). SKILL.nb couples code, NL-procedure, and intent-level execution alternatives with evidence-based decisions to release workflows, formalize or demote steps, and retire workflows.

Durable Agent Artifacts. As agents produce durable outputs, governing their creation, validation, and maintenance becomes critical. Systems typically manage this evolution through versioned execution logs and localized repair (ALAS (Geng & Chang, 2025)), git-like state checkpointing (AgentGit (Li et al., 2025)), or iterative notebook refinement (You et al., 2025; Elhashemy et al., 2025). At the runtime level, Atomix enforces strict transactional semantics for tool calls (Mohammadi et al., 2026). ReUseIt synthesizes pre/post conditions and fallback actions, evaluates conditions with LLM judgments of screenshots, and incorporates user guidance when recovery fails (Liu et al., 2025a). AgentBay and CowPilot support human intervention during agent execution (Piao et al., 2025; Huq et al., 2025), while SkillWeaver discovers and practices web skills to synthesize reusable APIs (Zheng et al., 2025). SKILL.nb combines deterministic runtime gates with asynchronous maintenance and trace-replay calibration of lifecycle thresholds.

SKILL.nb brings experience reuse, executable workflow steps, and versioned maintenance into a lifecycle governed by logged execution evidence. Its focus is the coupling of step-level execution choices with workflow release, formalization, demotion, and retirement.

3 SKILL.nb

We present SKILL.nb, an online–offline framework that uses execution evidence to govern reusable workflow artifacts. Its core mechanism is *selective formalization*: deciding when steps remain NL-guided, become executable, are demoted after instability, or contribute to workflow retirement. We

describe the artifact representation, lifecycle policy, adaptive thresholds, and runtime loop; complete algorithms are in Appendix A.6.

3.1 Versioned Workflow Artifacts

SKILL.nb stores reusable procedures in a versioned workflow repository $\mathcal{K}$, distinct from its execution-log repository (Appendix A.4). A workflow version $\mathcal{W}_v$, $v \in \{1, 2, \ldots\}$, and its steps are

$$\mathcal{W}_v = \langle I, X, S, M^W \rangle, \qquad s_i = \langle I_i, P_i, C_i, \Gamma_i, M_i^S \rangle, \qquad \Gamma_i = (\gamma_{i,\text{pre}}, \gamma_{i,\text{post}}).$$

Here I is the workflow intent, X the input schema and validation rules, $S = (s_1, \ldots, s_n)$ the ordered steps, and M^W workflow metadata. Each released step stores a local intent I_i, an NL procedure P_i, an optional executable realization C_i, executable pre/post gates Γ_i, and step metadata M_i^S.

Maintenance constructs gates through LLM selection of assertions grounded in recorded Playwright and OpenTelemetry evidence; the backend verifies the selected evidence programmatically. Gates execute deterministically without model calls or access to hidden benchmark labels (Appendix A.5.1). Metadata supports retrieval at workflow and step level: M^W indexes similar task flows, while M_i^S indexes analogous or specialized steps. In our implementation, Jupyter notebooks[1] interleave these components (Appendix A.1). We analyze the overhead of this representation in Appendix C.5. Although notebooks occupy more disk space, the agent can request a step's contents directly through the notebook interface, without reading or searching the entire file. In our controlled single-workflow comparison, this direct access reduces the estimated text supplied for step execution and repair relative to full-file Markdown activation, making more efficient use of the agent's context.

3.2 Selective Formalization as a Lifecycle Policy

Selective formalization governs both workflow maturity and step representation. For workflow version $\mathcal{W}_v$, let

$$y(\mathcal{W}_v) \in \{\texttt{provisional}, \texttt{released}, \texttt{retired}\}$$

denote whether it is under validation, available for retrieval and execution, or removed from active retrieval while retained for rollback and analysis. For each step s_i, let $z_i \in \{0, 1\}$ indicate whether a validated executable realization C_i is available. If $z_i = 0$, execution falls back to P_i or, when necessary, the bare intent I_i; $z = (z_1, \ldots, z_n)$ is the workflow formalization pattern.

Ideally, a lifecycle controller would choose (y, z) to trade offline maintenance cost against repeated runtime cost:

$$(y^\star, z^\star) = \arg\min_{y,z} \; C_{\text{maint}}(y, z \mid \mathcal{W}_v) + C_{\text{run}}(y, z \mid \mathcal{W}_v)$$
$$\text{s.t.} \quad J_{\text{perf}}(y, z \mid \mathcal{W}_v) \le J_{\text{perf}}^{\text{ref}} + \epsilon.$$

Here C_{maint} is cumulative offline maintenance inference cost, including distillation, validation, repair, promotion, demotion, and retirement review. We approximate each maintenance event e by token cost $c(e) = \text{tok}_{\text{in}}(e) + \text{tok}_{\text{out}}(e)$. The runtime term C_{run} includes online inference, fallbacks, retries, and browser/tool actions. The loss J_{perf} is estimated from offline replay validation and captures downstream failure, regressions on previously passing traces, or step-level degradation relative to $J_{\text{perf}}^{\text{ref}}$, with tolerance $\epsilon \ge 0$. Runtime gates do not access J_{perf} or hidden benchmark labels.

The full objective is not directly solved, since lifecycle actions change future repository state, trace distributions, and maintenance opportunities. SKILL.nb instead uses a restricted threshold policy π_θ, with $\theta = (\tau_{\text{create}}, \tau_{\text{form}}, \tau_{\text{demote}}, \tau_{\text{retire}})$, that emits create, form, demote, or retire actions from logged evidence. Create, form, and demote use count thresholds; retirement uses a normalized repair burden in $[0, 1]$. Threshold crossings trigger maintenance review, and repository state changes only after the corresponding artifact update passes validation.

The thresholds use two evidence types. *Trace-support evidence* governs workflow creation and step formalization. For request q, let $\mathcal{C}(q)$ be the related prior traces selected by retrieval before the lifecycle decision. The workflow-level count $|\mathcal{C}(q)|$ is compared with τ_{create}. For each step s_i, maintenance aligns traces in $\mathcal{C}(q)$ to the workflow sequence and counts validated execution segments supporting that step, giving n_i^{evidence}, which is compared with τ_{form}.

[1] https://jupyter.org/

Repair evidence governs demotion and retirement. Let m_i^{repair} be the number of accepted repairs affecting step s_i in version $\mathcal{W}_v$; only repairs passing maintenance validation are counted. If $m_i^{\text{repair}} \geq \tau_{\text{demote}}$, the executable realization for s_i becomes eligible for maintenance review for demotion to NL-guided execution. For retirement, repair count alone is too coarse, so we use a token-weighted burden. Let $\mathcal{R}_i(\mathcal{W}_v)$ be accepted repairs affecting step slot i along the lineage ending at $\mathcal{W}_v$, let $\mathcal{R}_{\text{cal}}$ be accepted repairs in calibration logs, and set $c_{\text{ref}} = \max_{e \in \mathcal{R}_{\text{cal}}} c(e)$. If $\mathcal{R}_{\text{cal}}$ is empty, automatic repair-burden retirement is deferred; otherwise,

$$\rho_{\text{repair}}(\mathcal{W}_v) = \frac{1}{|S|} \sum_{i=1}^{|S|} \min\left(1, \frac{\sum_{e \in \mathcal{R}_i(\mathcal{W}_v)} c(e)}{c_{\text{ref}}}\right) \in [0, 1].$$

This averages capped per-step repair burden, preventing a single repeatedly repaired step from dominating the retirement signal.

3.3 Adaptive Thresholds via Group Specialization

A single global θ is often too aggressive for sparse domains and too conservative for well-observed ones. SKILL.nb therefore treats groups as threshold-sharing units: workflow decisions (`create`, `retire`) use workflow groups $g^W \in \mathcal{G}^W$, while step decisions (`form`, `demote`) use step groups $g^S \in \mathcal{G}^S$. Groups are canonicalized from artifact metadata such as site family, task type, action type, and interface properties, excluding benchmark task IDs and hidden labels. When a statement applies to either workflow or step groups, we write $g \in \mathcal{G}_d$, where d is the lifecycle decision.

Let $\mathcal{L}_{\text{thr}}$ contain historical cases from completed executions in the same chronological task stream. Only cases available before a decision can inform it; no additional exploratory round or evaluator-labeled calibration split is used. A threshold-estimation case j for decision d contains the signal u_j, candidate lifecycle action, attributable maintenance token cost, and offline validation outcome; cases missing any of these fields are excluded. Let $\mathcal{D}_{g,d} \subseteq \mathcal{L}_{\text{thr}}$ be the usable cases for group g and decision d, with $n_{g,d} = |\mathcal{D}_{g,d}|$. The signal is $|\mathcal{C}(q)|$ for creation, n_i^{evidence} for formalization, m_i^{repair} for demotion, and $\rho_{\text{repair}}(\mathcal{W}_v)$ for retirement. Let $\mathcal{T}_d$ be the sorted unique signal values in the logs; threshold τ admits case j when $u_j \geq \tau$.

For candidate τ, $\hat{C}^{(d)}_{\text{maint}}(g, \tau)$ is the replay-estimated maintenance token cost for decision d on $\mathcal{D}_{g,d}$, omitting costs common to all thresholds. This optimizes maintenance compute rather than full C_{total}: runtime effects of suppressed actions are not reliably counterfactually observed; runtime efficiency, recovery, and regression are instead controlled through validation filters and measured downstream.

The validation-violation rate is the fraction of usable logged cases that both cross τ and exceed the offline validation tolerance:

$$\hat{V}^{(d)}(g, \tau) = k_{g,d}(\tau)/n_{g,d}, \qquad n_{g,d} > 0,$$

where $n_{g,d}$ counts all usable cases, and $k_{g,d}(\tau)$ counts admitted cases whose offline validation loss exceeds $J^{\text{ref}}_{\text{perf}} + \epsilon$. If $n_{g,d} = 0$, no group-specific violation rate is estimated. To avoid treating small groups as safe because they have few observed violations, a threshold is feasible only when the one-sided Wilson upper confidence bound is below the decision-specific budget $V^{(d)}_{\max}$:

$$\mathcal{F}_{g,d} = \left\{\tau \in \mathcal{T}_d : \text{WilsonUCB}_{1-\alpha}(k_{g,d}(\tau), n_{g,d}) \leq V^{(d)}_{\max}\right\}.$$

The pooled feasible set $\mathcal{F}^{\text{pool}}_d$ is computed analogously after pooling cases across groups; when nonempty, it supplies the default for groups with no usable cases. This replay filter uses only logged cases with observed outcomes; it does not simulate induced changes to later repository contents, traces, or repairs. Because thresholds are selected after sweeping many candidates, the Wilson bound is a conservative replay filter rather than a uniform post-selection or deployment-time safety guarantee.

If $\mathcal{F}^{\text{pool}}_d \neq \emptyset$, the pooled threshold is

$$\hat{\tau}^{\text{pool}}_d = \arg\min_{\tau \in \mathcal{F}^{\text{pool}}_d} \hat{C}^{(d)}_{\text{maint}}(\text{pool}, \tau).$$

If $\mathcal{F}_d^{\text{pool}} = \emptyset$, automatic thresholding for decision d is deferred to maintenance review. For a group with $n_{g,d} > 0$ and $\mathcal{F}_{g,d} \neq \emptyset$,

$$\hat{\tau}_d^g = \arg\min_{\tau \in \mathcal{F}_{g,d}} \hat{C}^{(d)}_{\text{maint}}(g, \tau).$$

For groups with usable cases, deployment blends group and pooled estimates only when both feasible sets are nonempty:

$$\tau_d^g = \begin{cases} \Pi_{\mathcal{F}_{g,d}}\left[\omega_{g,d}\hat{\tau}_d^g + (1-\omega_{g,d})\hat{\tau}_d^{\text{pool}}\right], & n_{g,d} > 0,\ \mathcal{F}_{g,d} \neq \emptyset,\ \mathcal{F}_d^{\text{pool}} \neq \emptyset, \\ \Pi_{\mathcal{F}_d^{\text{pool}}}\left[\hat{\tau}_d^{\text{pool}}\right], & n_{g,d} = 0,\ \mathcal{F}_d^{\text{pool}} \neq \emptyset, \\ \text{defer}_d, & \text{otherwise.} \end{cases}$$

Here $\omega_{g,d} = n_{g,d}/(n_{g,d} + n_0)$ and $\Pi_{\mathcal{F}}$ projects to the nearest replay-supported feasible value. The value defer_d means no automatic thresholded action is taken, and the case is routed to maintenance review. At runtime, π_θ compares each lifecycle signal with the deployed group threshold; crossings trigger review, not direct writes to $\mathcal{K}$. Appendix A.7 gives the Wilson formula, tie-breaking, empty-set handling, and full estimation procedure.

Artifact updates with a fixed base model. SKILL.nb keeps the base LLM fixed during runtime and offline maintenance. It responds to interface drift—changed DOM structure, selectors, or page flow (Choudhary et al., 2011; Leotta et al., 2014; 2016; Stocco et al., 2018)—by reviewing and revising workflow artifacts. Release, formalization, demotion, and retirement use trace support and accepted repairs, with replay validation filtering candidate thresholds. Appendix A.7.3 compares this artifact-level control with end-to-end RL.

3.4 Runtime Execution and Maintenance Loop

Runtime executes the latest applicable released workflow. Otherwise, the agent performs the task while recording a provisional workflow $\hat{\mathcal{W}}$ and telemetry, using observed environment changes to assess progress. Maintenance constructs missing gates afterward. It maintains temporary per-run memory $\mathcal{M}$ alongside the authoritative repository $\mathcal{K}$; $\mathcal{M}$ is mutable and non-authoritative, storing transient observations, local repairs, and provisional routines.

Steps with established gates follow $C_i \to P_i \to I_i$, rechecking the pre-gate on current state before each attempt and after local drift repair; completion requires a passing post-gate. Failed attempts with observed potentially mutating events terminate the run; others permit bounded fallback or recovery. This trace-based rule does not guarantee exactly-once effects. Unresolved traces go to maintenance. Only offline-validated changes enter $\mathcal{K}$, and only accepted repairs update repair evidence. Appendix A.6 gives both algorithms and their scope.

Offline maintenance closes the loop. Each execution is distilled into non-authoritative temporary evidence in $\mathcal{M}$ and, when warranted, proposals for durable updates. This yields a retrieve $\to$ execute $\to$ distill $\to$ promote loop in which runtime proposes changes while maintenance agents, counted in maintenance cost, verify, refactor, de-duplicate, and promote artifacts into new repository versions. The thresholds from §3.3 govern workflow release or retirement and step formalization or demotion.

4 Experiments

We evaluate SKILL.nb along three axes: fresh-start task performance on WebArena-Verified and directional transfer on Mind2Web (§4.1), repeated-use lifecycle reliability (§4.2), and real application-version drift (§4.3). Supporting these experiments, we show detailed component ablations (Appendix C.1) and threshold specialization (Appendix C.2).

Baselines. We compare SKILL.nb against baselines targeting different capabilities. We use CodeAct (Wang et al., 2024b), $\text{AWM}_{\text{online}}$ (Wang et al., 2025b), and ReasoningBank (Ouyang et al., 2025), representing executable action generation, workflow memory, and retrieved reasoning memory, respectively.

Benchmarks. To align with prior workflow- and reasoning-memory evaluations (Wang et al., 2025b; Ouyang et al., 2025), we evaluate on WebArena-Verified (El hattami et al., 2025) and

Mind2Web (Deng et al., 2023). WebArena-Verified provides 812 tasks across five self-hosted websites. We use its 258-task hard subset for component and threshold ablations. Mind2Web is used for single-round generalization on the cross-task, cross-website, and cross-domain test splits.

Evaluation Protocol. All methods are re-evaluated in a shared harness with method-specific inference settings and native update paths. WebArena-Verified reports 95% Wilson CIs and uses a two-sided continuity-corrected McNemar test for the main paired success comparison; Mind2Web reports point estimates because its metrics mix macro-averaged and task-level quantities. Each experiment below provides specific experimental setup and full protocol details are in Appendix B.1.

4.1 Benchmark Performance and Generalization

WebArena-Verified. We report single-round performance on all 812 WebArena-Verified tasks. Each method starts without task-specific persistent state and builds any repository or memory online during that round. The primary metric is task success rate (SR). Table 1 reports overall and per-website SR. We do not compare raw step counts because methods introduce different step types by design.

Table 1: SKILL.nb achieves 53.7% success on WebArena-Verified, 3.9 points above the next-best baseline in the evaluated online round (reported paired $p = 0.029$). Overall SR is the task-weighted success rate across all 812 tasks (95% Wilson CI in brackets). Site-specific columns report SR point estimates for each domain. All methods start without preloaded workflow or memory states.

Method	SR (95% CI) ↑	Shopping	Admin	Reddit	GitLab	Maps	Multi
CodeAct	38.3 [35.0, 41.7]	39.0	44.5	55.7	31.2	33.9	10.2
AWM_{online}	46.4 [43.0, 49.9]	48.1	49.3	68.9	40.0	45.3	5.7
ReasoningBank	49.8 [46.3, 53.2]	**52.1**	55.8	68.9	45.2	42.1	11.3
SKILL.nb (ours)	**53.7 [50.3, 57.1]**	52.0	**56.2**	**70.3**	**54.4**	**50.8**	**17.5**

SKILL.nb achieves the highest SR at 53.7%, outperforming ReasoningBank by 3.9 percentage points ($p = 0.029$), AWM_{online} by 7.3 points, and CodeAct by 15.4 points. The largest site-level gains appear on GitLab (+9.2 points over the next-best baseline) and Maps (+5.5 points). This online round permits learning across requests and does not measure a first-request-only setting.

Cold-start setting. In separate runs resetting persistent state before every task, SKILL.nb solves 396/812 (48.8%) versus ReasoningBank's 377/812 (46.4%), at 50k versus 41k recorded inference tokens per success (approximately 22% more). This measures the complete within-task stacks without accumulated state; no paired significance or isolated lifecycle benefit is claimed. Appendix C.4 gives the protocol and results.

Table 2: SKILL.nb achieves the best overall performance on Mind2Web, leading on all metrics in the cross-website and cross-domain splits. Results provide directional evidence of transfer, especially at the step level. We show element accuracy (EA, ↑), action F_1 (AF_1, ↑), step success rate (SSR, ↑), and task success rate (SR, ↑). EA, AF_1 are macro-averaged; SSR, SR are micro-averaged.

	Cross-Task (252)				Cross-Website (177)				Cross-Domain (912)			
Method	EA	AF_1	SSR	SR	EA	AF_1	SSR	SR	EA	AF_1	SSR	SR
CodeAct	46.0	59.1	40.3	3.3	39.8	45.1	31.7	1.7	35.8	37.9	31.9	1.0
AWM_{online}	50.0	56.4	43.6	4.0	42.1	45.1	31.6	1.6	40.9	36.3	35.5	1.5
ReasoningBank	**52.1**	60.4	44.9	**4.8**	44.3	52.6	34.9	3.1	40.6	41.3	36.6	1.6
SKILL.nb (ours)	52.0	**63.2**	**45.0**	4.7	**46.8**	**54.3**	**38.1**	**4.0**	**44.9**	**46.0**	**39.7**	**1.9**

Mind2Web. We evaluate generalization on Mind2Web in a single benchmark round, starting without task-specific persistent state and allowing online workflow or memory accumulation according to each method's design (Table 2; Appendix B.1). Absolute task success remains low across all methods, but SKILL.nb achieves the best values across all four metrics in the cross-website and cross-domain settings. On the cross-task split, SKILL.nb closely matches ReasoningBank on task success but leads by 2.8 percentage points on action F_1. These point estimates provide directional evidence of transfer, especially at the step level. Shared high-level intents across websites and recurring interactions such as search and form filling across domains offer plausible explanations for this pattern.

4.2 LIFECYCLE DYNAMICS: REUSE CONSISTENCY, REPAIR, AND REGRESSION

Beyond initial success, we investigate whether persistent workflow or memory states benefit agents on recurring tasks. We assess repeated execution and native repair. The first analysis measures round-level reuse (Figure 1(a)): methods execute five WebArena-Verified rounds, carrying persistent states across iterations while per-task environment states and transient contexts reset. Each round applies a uniform perturbation protocol across all methods: tasks are reshuffled, starting URLs are varied, and intent templates are paraphrased while preserving core slots and intents. We also measure artifact reuse and repair under perturbation. Initially successful states are snapshotted and re-executed three times without updates to measure reuse consistency (Figure 1(b)). Recovery and update-induced regression are measured through each method's native update path (Figure 1(c)). Appendix B.1 gives the full protocol.

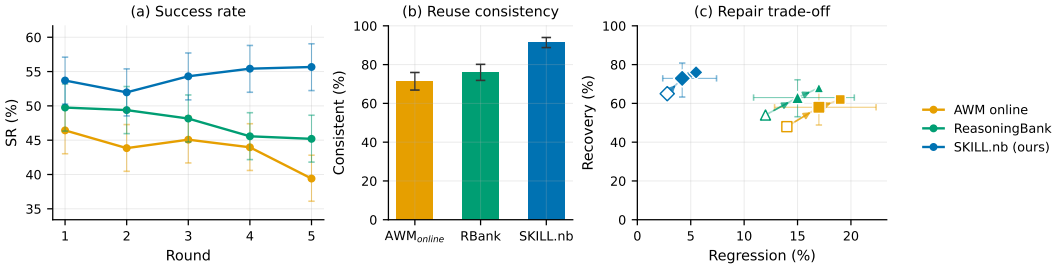


Figure 1: SKILL.nb improves task success over repeated rounds, maintains the highest reuse consistency, and combines higher repair recovery with lower regression than the compared baselines. (a) Task success over five perturbed rounds. (b) Reuse consistency: fraction of workflows surviving three perturbed re-executions without updates. (c) Recovery vs. regression under each method's native update path. Emphasized markers denote repair budget 2 (Table 11). Error bars indicate 95% Wilson CIs.

SKILL.nb is the only method to improve over repeated rounds, rising from 53.7% to 55.7% by round 5, while baselines decline by 4.6–7.0 percentage points (Figure 1(a)). In the reuse test, SKILL.nb maintains 91.7% reuse consistency, above ReasoningBank (76.2%) or AWM$_{\text{online}}$ (71.6%; Figure 1(b)). When artifacts fail and enter native update paths, SKILL.nb combines higher recovery with lower regression (72.9%/4.2%), outperforming ReasoningBank (63.0%/15.0%) and AWM$_{\text{online}}$ (58.0%/17.0%) at budget 2 (Figure 1(c)). These results show improved reuse consistency and lower update-induced regression under the evaluated perturbation and native-repair protocols.

Mechanism and overhead diagnostics. Table 3 shows component ablations on the WebArena-Verified hard subset. The full system has the highest point SR and lowest SKILL.nb-internal maintenance/update token cost per success. Removing executable formalization or disabling fallback lowers success, while removing gates or lifecycle cleanup increases update-induced regression. In a separate three-round threshold-policy ablation (Figure 13), SKILL.nb reaches 38.4% success with 3.3% regression by round 3, compared with 27.1% success and 22.0% regression for loose fixed thresholds. Appendix C.2 reports the accompanying maintenance-token comparison. SKILL.nb records 52k inference tokens per success in round 1 and 36k in round 5; its five-round aggregate exceeds both baselines. Full ablations, threshold curves, and cost curves are in Appendices C.1, C.2, and B.3.

Table 3: Main mechanism ablation summary on the 258-task WebArena-Verified hard subset. No cleanup disables demotion and retirement together. SR and regression are percentages; token cost is SKILL.nb-internal maintenance/update tokens per success. Full diagnostics are in Appendix C.1.

Variant	SR (%) ↑	Regression (%) ↓	Tok./succ. ↓
SKILL.nb	**38.4**	**3.3**	**36k**
NL-only	33.3	8.8	57k
Code-only	31.0	14.7	42k
No gates	32.6	18.6	45k
No cleanup	34.1	12.4	47k

4.3 WORKFLOW REUSE UNDER GITLAB VERSION DRIFT

We next test whether persistent state remains useful under real interface drift. We migrate all 180 single-site GitLab tasks from WebArena-Verified from GitLab 15.7 to GitLab 16.11 as an intermediate target and GitLab 18.9 as a longer-horizon target. Appendix Figure 12 and Table 13 illustrate the corresponding UI and DOM drift. Unlike synthetic perturbations, these updates change DOM structure, selectors, and page flow while preserving user intent. Each method is evaluated in five conditions: fresh-start runs on GitLab 15.7, 16.11, and 18.9, plus source-snapshot reuse on GitLab 16.11 and 18.9 using the repository or memory built on GitLab 15.7. Before each reuse task, we restore the same source snapshot and discard durable updates afterward. Within the task, SKILL.nb's gates, code → NL procedure → intent fallback, and local repair remain active. This tests recovery from the same inherited state at each request; persistent updates do not accumulate across target-version tasks. Full protocol details are in Appendix B.4.

Fresh-start success is higher on both target deployments than on GitLab 15.7 for all three evaluated methods. Qualitative log inspection suggests that newer GitLab versions expose more stable DOM locators for the Playwright-based harness. They may also better match recent LLM web-navigation related pre-training knowledge.

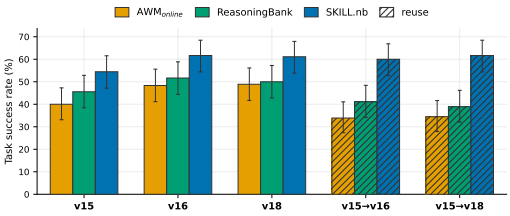


Figure 2: Fresh-start and source-snapshot reuse performance under real GitLab version drift. The x-axis shows fresh-start runs on GitLab 15.7, 16.11, and 18.9, followed by source-snapshot reuse from 15.7→16.11 and 15.7→18.9. Within each condition, orange, green, and blue bars denote AWM$_{online}$, ReasoningBank, and SKILL.nb. Hatched bars denote source-snapshot reuse with native within-task recovery. Error bars indicate 95% Wilson CIs.

Persistent-memory baselines suffer negative transfer from frozen source-version state. On GitLab 18.9, AWM$_{online}$ drops from 88/180 fresh-start successes to 62/180 frozen-state successes (48.9% to 34.4%), a 14.4-point degradation. ReasoningBank similarly drops from 90/180 to 70/180 successes (50.0% to 38.9%), an 11.1-point degradation. Qualitative inspection suggests that frozen memories often retain interface-specific assumptions, such as brittle click identifiers or page-geometry-dependent procedures, that no longer match the migrated DOM. In contrast, SKILL.nb preserves target-version performance under frozen-state reuse. On GitLab 18.9, it obtains 110/180 fresh-start successes and 111/180 frozen-state successes (61.1% vs. 61.7%). The same pattern holds on GitLab 16.11, where frozen-state reuse reaches 108/180 successes compared with 111/180 fresh-start successes. This result is consistent with gate-conditioned selective formalization: when an executable realization no longer satisfies its pre/postcondition gates, runtime can fall back from code to the preserved NL procedure or step intent. Thus, under real GitLab version drift, SKILL.nb avoids the old-state degradation observed in persistent-memory baselines.

5 CONCLUSION

We presented SKILL.nb, a lifecycle-governance framework for durable web-agent workflow artifacts. Its core mechanism, *selective formalization*, uses execution-grounded evidence to decide when workflows are released or retired, and when steps remain NL-guided, become executable, or are demoted after repair. Versioned notebooks provide the governed artifact: they bind NL procedures, executable realizations, validation gates, and maintenance history into a reusable object. Across WebArena-Verified, Mind2Web, and GitLab version drift, SKILL.nb improves single-round task performance, repeated-use reliability, repair recovery, and robustness to stale persistent state. These results suggest that durable agent artifacts should be treated not only as memories, but as lifecycle-managed objects whose promotion, repair, and retirement are governed by execution evidence. Limitations are discussed in Appendix A.8.

AI USE STATEMENT

We used generative AI tools to assist with manuscript editing, including checks for typographical and grammatical errors, and to analyze experiment logs, including agent traces and environment logs, to identify recurring patterns. These patterns served as input to subsequent manual analysis by the authors. The use of language-model agents within the evaluated systems is described in Sections 3–4 and Appendix B.1.

REFERENCES

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready AI agents with scalable long-term memory, 2025. URL https://arxiv.org/abs/2504.19413.

Shauvik Roy Choudhary, Dan Zhao, Husayn Versee, and Alessandro Orso. WATER: Web application test repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, 2011.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2Web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114, December 2023. URL https://arxiv.org/abs/2306.06070.

Amine El hattami, Megh Thakkar, Nicolas Chapados, and Christopher Pal. WebArena Verified: Reliable evaluation for web agents. In *Workshop on Scaling Environments for Agents at NeurIPS*, 2025. URL https://openreview.net/forum?id=94tlGxmqkN.

Hanya Elhashemy, Youssef Lotfy, and Yongjian Tang. Bridging the prototype-production gap: A multi-agent system for notebooks transformation, 2025. URL https://arxiv.org/abs/2511.07257.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. MemP: Exploring agent procedural memory, 2025. URL https://arxiv.org/abs/2508.06433.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Qihan Ren, Yiran Wu, Hongru WANG, Han Xiao, Yuhang Zhou, Shaokun Zhang, Jiayi Zhang, Jinyu Xiang, Yixiong Fang, Qiwen Zhao, Dongrui Liu, Cheng Qian, Zhenhailong Wang, Minda Hu, Huazheng Wang, Qingyun Wu, Heng Ji, and Mengdi Wang. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *Transactions on Machine Learning Research*, 2026. ISSN 2835-8856. URL https://openreview.net/forum?id=CTr3bovS5F. Survey Certification.

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.

Longling Geng and Edward Y Chang. ALAS: Transactional and dynamic multi-agent llm planning. *arXiv preprint arXiv:2511.03094*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan

Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633–638, September 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. HiAgent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 32779–32798, Vienna, Austria, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.1575. URL https://aclanthology.org/2025.acl-long.1575/.

Faria Huq, Zora Zhiruo Wang, Frank F. Xu, Tianyue Ou, Shuyan Zhou, Jeffrey P. Bigham, and Graham Neubig. Cowpilot: A framework for autonomous and human-agent collaborative web navigation. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (System Demonstrations)*, pp. 163–172, Albuquerque, New Mexico, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.naacl-demo.17. URL https://aclanthology.org/2025.naacl-demo.17/.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V. Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, Yuling Gu, Saumya Malik, Victoria Graf, Jena D. Hwang, Jiangjiang Yang, Ronan Le Bras, Oyvind Tafjord, Chris Wilhelm, Luca Soldaini, Noah A. Smith, Yizhong Wang, Pradeep Dasigi, and Hannaneh Hajishirzi. Tulu 3: Pushing frontiers in open language model post-training, 2025. URL https://arxiv.org/abs/2411.15124.

Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Visual vs. DOM-based web locators: An empirical study. In *International Conference on Web Engineering*, 2014.

Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. ROBULA+: An algorithm for generating robust XPath locators for web testing. *Journal of Software: Evolution and Process*, 28: 177–204, 2016. doi: 10.1002/smr.1771. URL https://doi.org/10.1002/smr.1771.

Ido Levy, Ben Wiesel, Sami Marreed, Alon Oved, Avi Yaeli, and Segev Shlomov. ST-WebAgentBench: A Benchmark for Evaluating Safety and Trustworthiness in Web Agents, May 2025. URL https://arxiv.org/abs/2410.06703v4. arXiv:2410.06703 [cs].

Yang Li, Siqi Ping, Xiyu Chen, Xiaojian Qi, Zigan Wang, Ye Luo, and Xiaowei Zhang. AgentGit: A version control framework for reliable and scalable llm-powered multi-agent systems. *arXiv preprint arXiv:2511.00628*, 2025.

Xuechen Liang, Yangfan He, Yinghui Xia, Xinyuan Song, Jianhui Wang, Meiling Tao, Li Sun, Xinhang Yuan, Jiayi Su, Keqin Li, Jiaqi Chen, Jinsong Yang, Siyuan Chen, and Tianyu Shi. Self-evolving agents with reflective and memory-augmented abilities, 2024. URL https://arxiv.org/abs/2409.00872.

Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*, 2024.

Yimeng Liu, Misha Sra, Jeevana Priya Inala, and Chenglong Wang. ReUseIt: Synthesizing reusable ai agent workflows for web automation. *arXiv preprint arXiv:2510.14308*, 2025a.

Yitao Liu, Chenglei Si, Karthik R. Narasimhan, and Shunyu Yao. Contextual experience replay for self-improvement of language agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14179–14198, Vienna, Austria, 2025b. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.694. URL https://aclanthology.org/2025.acl-long.694/.

Bodhisattwa Prasad Majumder, Bhavana Dalvi Mishra, Peter Jansen, Oyvind Tafjord, Niket Tandon, Li Zhang, Chris Callison-Burch, and Peter Clark. CLIN: A continually learning language agent for rapid task adaptation and generalization, 2023. URL https://arxiv.org/abs/2310.10134.

Bardia Mohammadi, Nearchos Potamitis, Lars Klein, Akhil Arora, and Laurent Bindschaedler. Atomix: Timely, transactional tool use for reliable agentic workflows. *arXiv preprint arXiv:2602.14849*, 2026.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. 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, 2025.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems, 2023. URL https://arxiv.org/abs/2310.08560.

Yun Piao, Hongbo Min, Hang Su, Leilei Zhang, Lei Wang, Yue Yin, Xiao Wu, Zhejing Xu, Liwei Qu, Hang Li, Xinxin Zeng, Wei Tian, Fei Yu, Xiaowei Li, Jiayi Jiang, Tongxu Liu, Hao Tian, Yufei Que, Xiaobing Tu, Bing Suo, Yuebing Li, Xiangting Chen, Zeen Zhao, Jiaming Tang, Wei Huang, Xuguang Li, Jing Zhao, Jin Li, Jie Shen, Jinkui Ren, and Xiantao Zhang. AgentBay: A hybrid interaction sandbox for seamless human-AI intervention in agentic systems, 2025. URL https://arxiv.org/abs/2512.04367.

Cheng Qian, Shihao Liang, Yujia Qin, Yining Ye, Xin Cong, Yankai Lin, Yesai Wu, Zhiyuan Liu, and Maosong Sun. Investigate-consolidate-exploit: A general strategy for inter-task agent self-evolution, 2024. URL https://arxiv.org/abs/2401.13996.

Hanif Ur Rahman, Asaad Alzayed, Muhammad Ismail Mohmand, Abdullah M. Albarrak, and Sultan Noman Qasem. Application maintenance offshoring using HCI based framework and simple multi attribute rating technique (SMART). *IEEE Access*, 11:107068–107084, 2023. doi: 10.1109/ACCESS.2023.3320941.

Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. Visual web test repair. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 503–514. ACM, 2018. doi: 10.1145/3236024.3236063. URL https://doi.org/10.1145/3236024.3236063.

Hongjin Su, Ruoxi Sun, Jinsung Yoon, Pengcheng Yin, Tao Yu, and Sercan O. Arik. Learn-by-interact: A data-centric framework for self-adaptive agents in realistic environments. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=3UKOzGWCVY.

Zhen Tan, Jun Yan, I-Hung Hsu, Rujun Han, Zifeng Wang, Long Le, Yiwen Song, Yanfei Chen, Hamid Palangi, George Lee, Anand Rajan Iyer, Tianlong Chen, Huan Liu, Chen-Yu Lee, and Tomas Pfister. In prospect and retrospect: Reflective memory management for long-term personalized dialogue agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8416–8439, Vienna, Austria, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.413. URL https://aclanthology.org/2025.acl-long.413/.

Xiangru Tang, Tianyu Hu, Muyang Ye, Yanjun Shao, Xunjian Yin, Siru Ouyang, Wangchunshu Zhou, Pan Lu, Zhuosheng Zhang, Yilun Zhao, Arman Cohan, and Mark Gerstein. ChemAgent: Self-updating library in large language models improves chemical reasoning. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/fa7f64b45970e6a7f8824781e7e01501-Paper-Conference.pdf.

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*, March 2024a. ISSN 2835-8856. URL https://openreview.net/forum?id=ehfRiF0R3a.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *International Conference on Machine Learning*, 2024b.

Zhiruo Wang, Graham Neubig, and Daniel Fried. TroVE: Inducing verifiable and efficient toolboxes for solving programmatic tasks. In *Forty-First International Conference on Machine Learning*, 2024c. URL https://openreview.net/forum?id=DCNCwaMJjI.

Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. Inducing Programmatic Skills for Agentic Tasks. In *Proceedings of the Conference on Language Modeling (COLM 2025)*, 2025a. URL https://openreview.net/forum?id=lsAY6fWsog.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 63897–63911. PMLR, 2025b. URL https://proceedings.mlr.press/v267/wang25bx.html.

Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. OS-Copilot: Towards generalist computer agents with self-improvement. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*, 2024. URL https://arxiv.org/abs/2402.07456.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-MEM: Agentic memory for LLM agents, 2025. URL https://arxiv.org/abs/2502.12110.

Ziming You, Yumiao Zhang, Dexuan Xu, Yiwei Lou, Yandong Yan, Wei Wang, Huamin Zhang, and Yu Huang. DatawiseAgent: A notebook-centric LLM agent framework for adaptive and robust data science automation. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 1099–1123, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.58. URL https://aclanthology.org/2025.emnlp-main.58/.

Mingtian Zhang, Yu Tang, and PageIndex Team. PageIndex: Next-generation vectorless, reasoning-based RAG. *PageIndex Blog*, September 2025. https://pageindex.ai/blog/pageindex-intro.

Zeyu Zhang, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Quanyu Dai, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model based agents, 2024. URL https://arxiv.org/abs/2404.13501.

Zhuosheng Zhang and Aston Zhang. You only look at screens: Multimodal chain-of-action agents, June 2024.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17):19632–19642, 2024. doi: 10.1609/aaai.v38i17.29936. URL https://doi.org/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, 2025. URL https://arxiv.org/abs/2504.07079.

Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. Synapse: Trajectory-as-exemplar prompting with memory for computer control. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/52f050499cf82fa8efb588e263f6f3a7-Abstract-Conference.html.

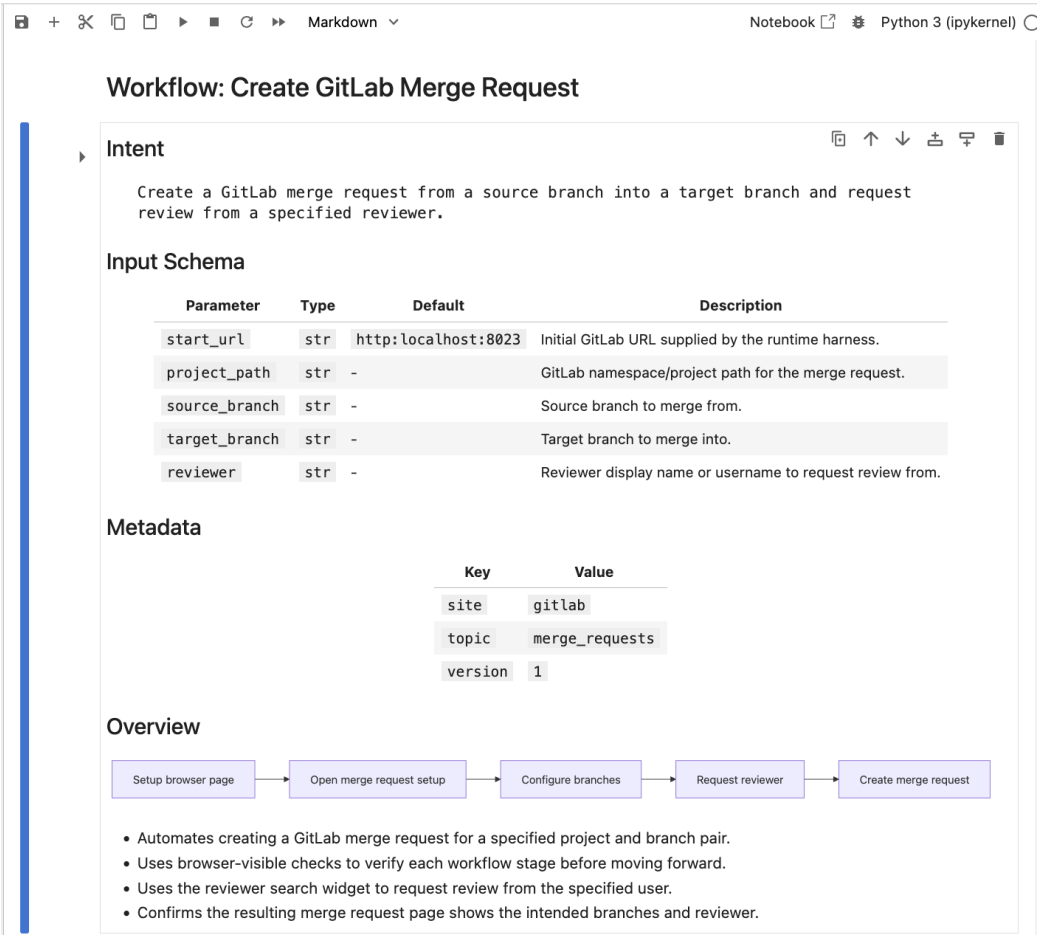
Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. MemoryBank: Enhancing large language models with long-term memory. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17):19724–19731, March 2024. ISSN 2374-3468. doi: 10.1609/aaai.v38i17.29946.

A METHOD DETAILS

A.1 FROM PROVISIONAL TRACE TO RELEASED WORKFLOW ARTIFACT



(a) Cell 1: provisional header cell with task-specific values.



(b) Cell 1: released header cell with reusable inputs and metadata.

Figure 3: Header-cell generalization for the GitLab merge-request lifecycle example. The provisional notebook records the concrete task request in Cell 1, while the released artifact keeps the reusable workflow description, input schema, and metadata so future requests can instantiate the same procedure with different branches, projects, and reviewers.

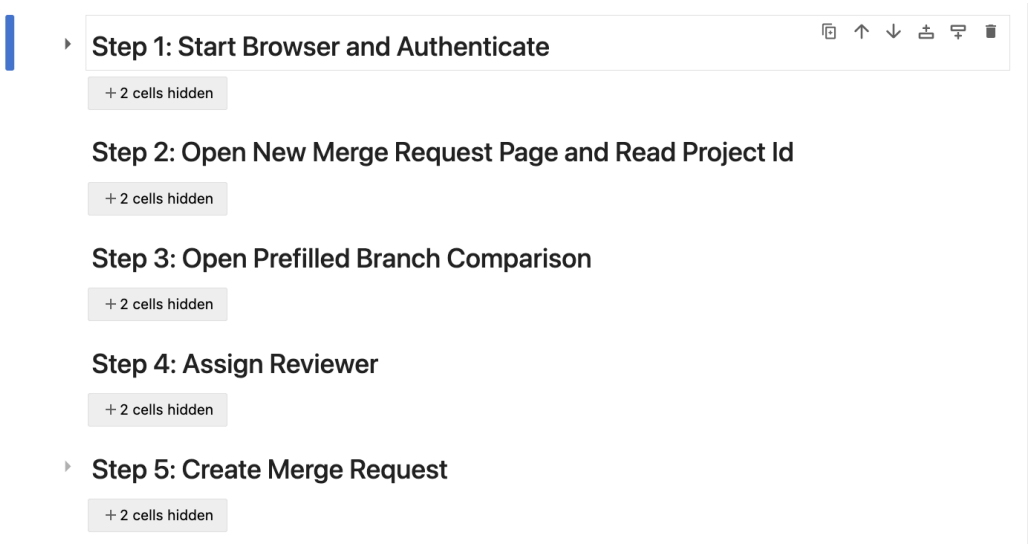
This appendix illustrates how a task-specific provisional notebook is converted into a reusable released SKILL.nb workflow artifact. The example uses a GitLab merge-request task to show how concrete task values are lifted into an input schema, how notebook cells are aligned to workflow steps, and how execution evidence supports validation and promotion. Runtime gates use browser-observable

state and task-provided expected values; notebook outputs, logs, screenshots, and cell status are used by offline maintenance during artifact review. The example does not use hidden benchmark labels or evaluator outputs.

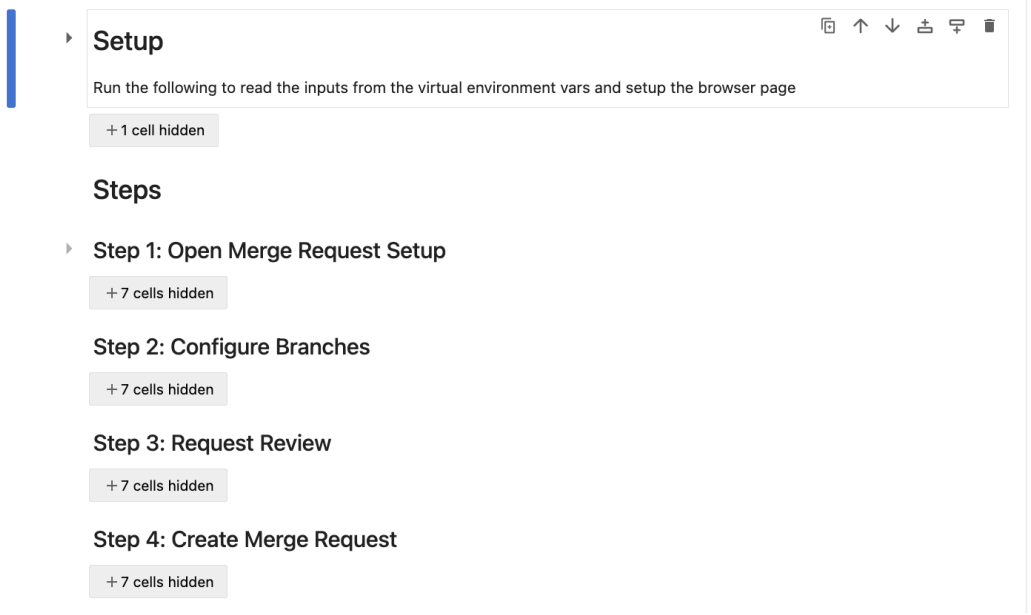
The concrete example is Task 668, a GitLab task starting from `http://localhost:8023`. The user request is to submit a merge request for source branch `redesign` in project `a11yproject.com`, merge it into the `main` branch, and assign `Roshan Jossy` as the reviewer. This task instance supplies concrete values for the source branch, target branch, project, and reviewer. During maintenance, these values are lifted into an input schema, while the task-specific request is rewritten as a generic merge-request workflow intent. This keeps the artifact reusable without storing the benchmark task ID as part of the executable workflow.

Figure 3 shows the first promotion transformation. The task-specific request supplies concrete inputs: source branch `redesign`, target branch `main`, source project `a11yproject.com`, and reviewer `Roshan Jossy`. Maintenance distills this instance into a reusable header cell that records the workflow description, input schema X, and workflow metadata M^W for retrieval.

After the provisional workflow executes, maintenance uses notebook evidence to align traces to the step cells in Figure 4, then expands individual stages as in Figure 5 to decide which steps should be formalized as executable cells with gates. In this example, trace evidence supports promotion from $y(\mathcal{W}_v) = \texttt{provisional}$ to $y(\mathcal{W}_v) = \texttt{released}$, while validated executable cells change the corresponding step indicators from $z_i = 0$ to $z_i = 1$. Each formalized step stores a local intent I_i, natural-language procedure P_i, executable realization C_i, executable pre/post gates Γ_i, and metadata M_i^S. The gates can check the current GitLab context, branch fields, reviewer selection, navigation state, and successful form submission. If later repair evidence shows that the workflow has become too costly or unstable, maintenance can move the same workflow lineage to $y(\mathcal{W}_v) = \texttt{retired}$ while retaining it for rollback and analysis. Because creating a merge request is non-idempotent, maintenance validation controls promotion rather than blindly rerunning the same submission; Appendix A.4.5 details mock-based validation.

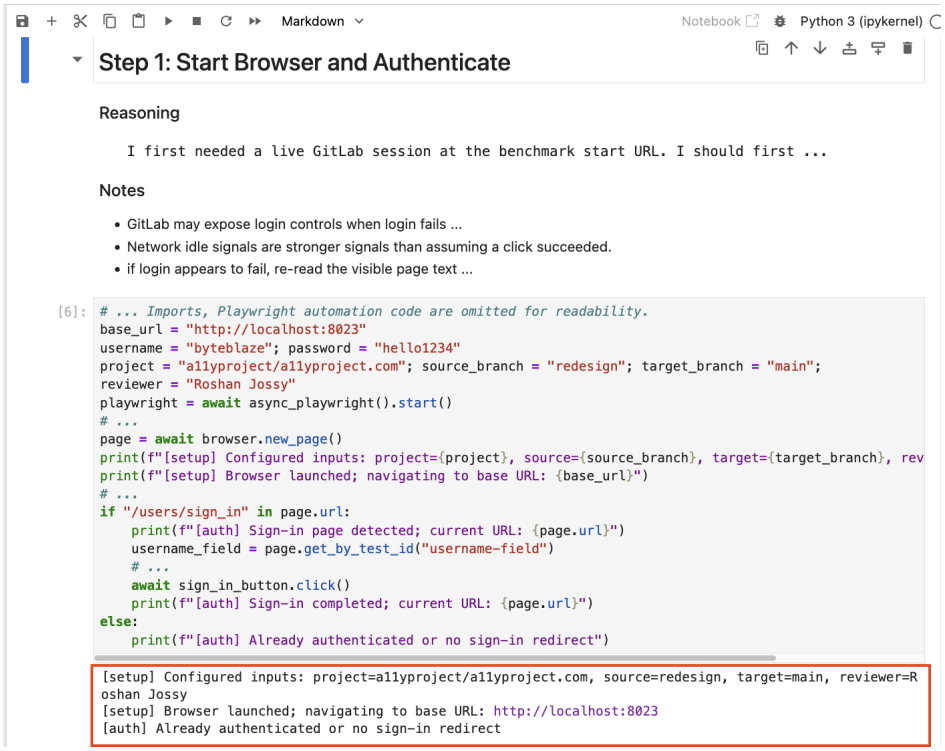


(a) Cells 2–N: provisional workflow steps.

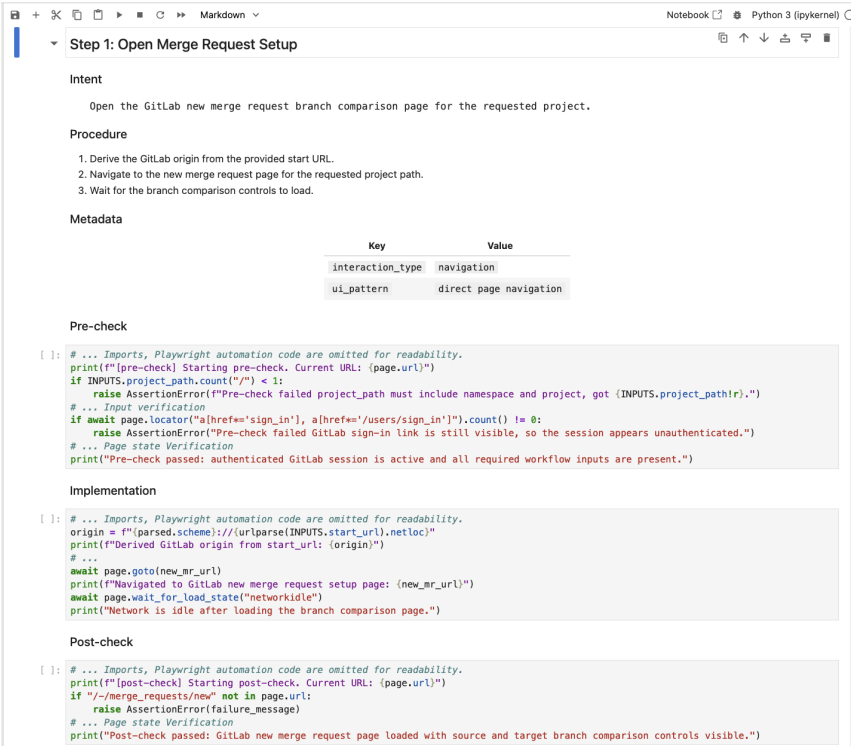


(b) Cells 2–N: released workflow steps with executable gates.

Figure 4: Step-cell transformation for the GitLab merge-request lifecycle example. Cells 2–N contain the executable workflow steps: setup, browser actions, checks, and submission. The released artifact preserves the reusable step structure while replacing task-specific traces with parameterized inputs and validation gates.



(a) Cells 2–3: provisional Step 1 with setup, checks, and red-boxed execution log.



(b) Cells 2–8: released Step 1 with parameterized setup and gates.

Figure 5: Expanded Step 1 transformation for the GitLab merge-request lifecycle example. The provisional step in (a), from the provisional notebook, is formalized into (b), from the released notebook, as $s_1 = \langle I_1, P_1, C_1, \Gamma_1, M_1^S \rangle$ with intent, procedure, executable setup, gates, and metadata.

A.2 Notebook Evidence and Debugging Affordances

The previous subsection used Task 668 to illustrate artifact promotion. The next figures show additional notebook affordances used by maintenance across workflows: cell-attached screenshots, interactive debugging state, localized failure evidence, and heterogeneous execution cells.

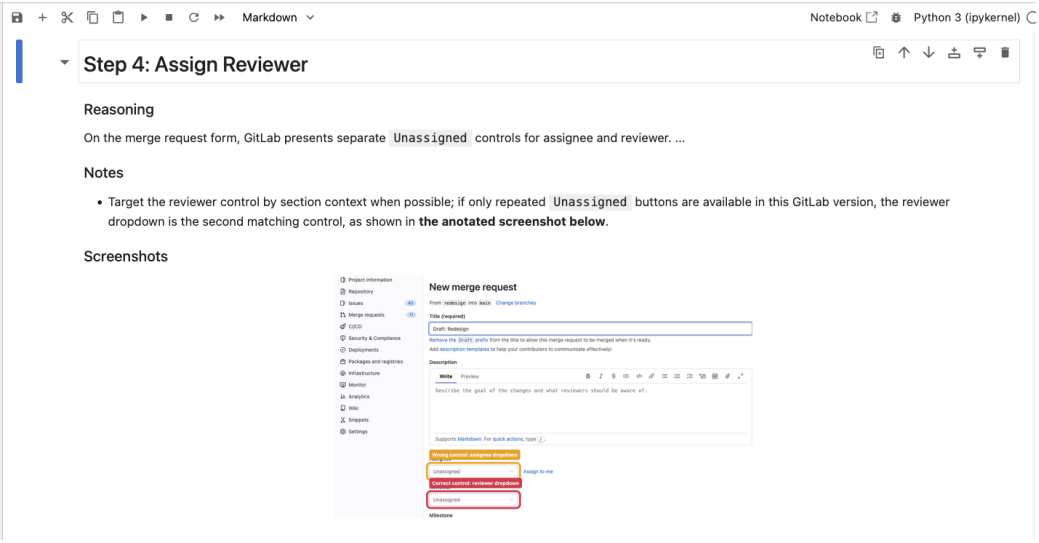


Figure 6: Cell-attached screenshot evidence for dynamic form handling. When provisional creation or execution encounters ambiguous controls, such as repeated `Unassigned` dropdowns for assignee and reviewer, the agent attaches the observed UI to the relevant cell. The embedded screenshot is cropped for readability; the actual notebook can link to the full-page screenshot used by maintenance to formalize reusable gates.

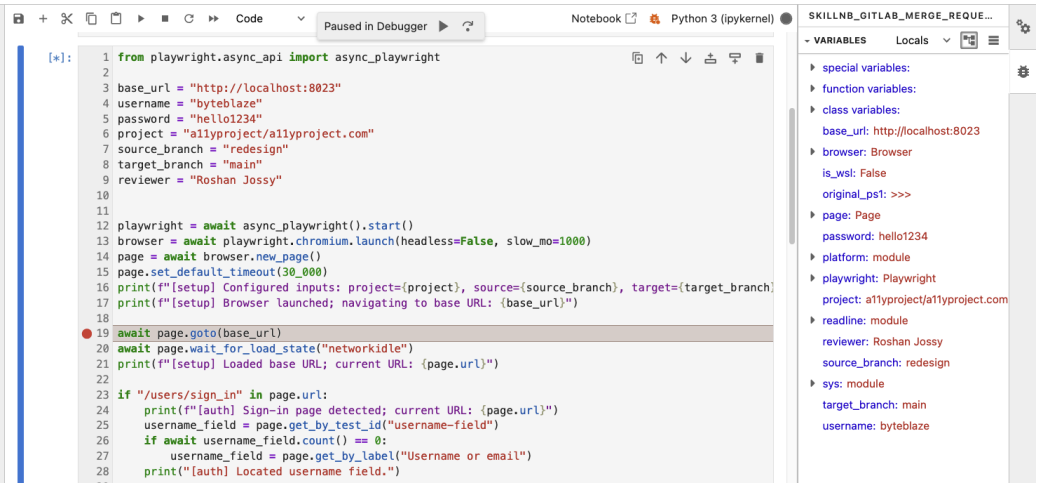


Figure 7: Interactive debugging support from the notebook representation. Because workflow realizations live in executable Jupyter cells, an agent or maintenance process can pause at a breakpoint, step through browser automation code, inspect task inputs and Playwright page state, and turn observed failures into more reliable executable gates.

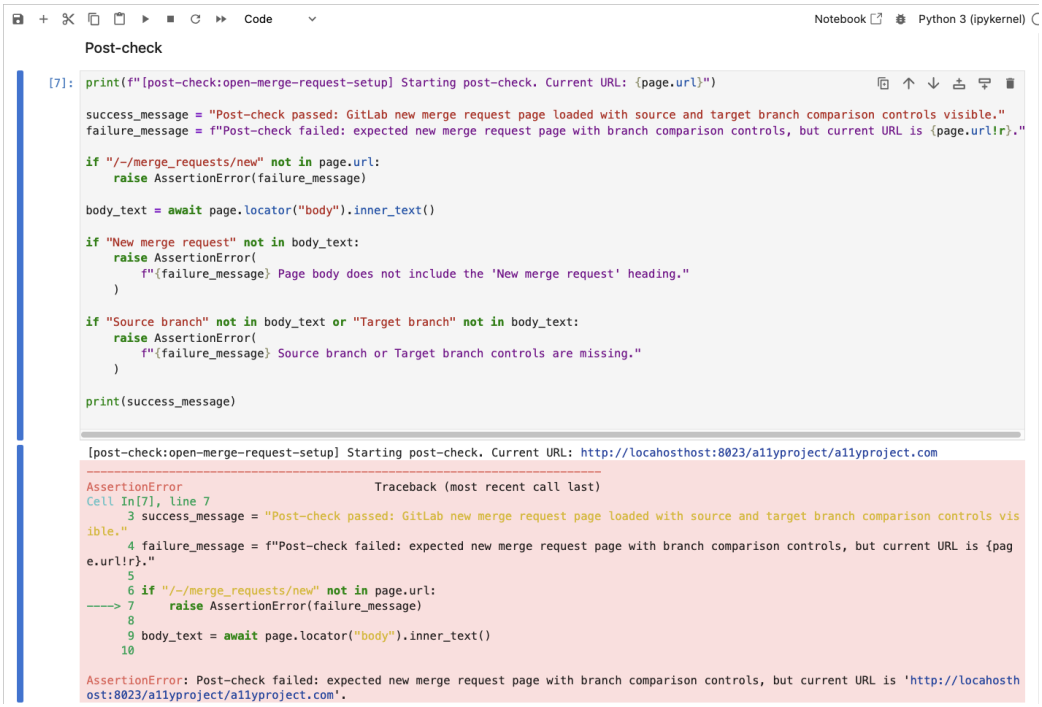


Figure 8: Cell-local failure evidence in a provisional notebook. When a gate fails, Jupyter stores the printed context, exception, and traceback directly under the failing cell, so an agent can retrieve the relevant code, state, and error in one localized artifact. A multi-component logging system could provide similar information, but would add logging infrastructure and synchronization points outside the workflow artifact.

Although the main walkthrough uses Task 668, the notebook representation also supports heterogeneous execution cells. Figure 9 shows a separate Task 784 example using Bash and Python cells in the same artifact.

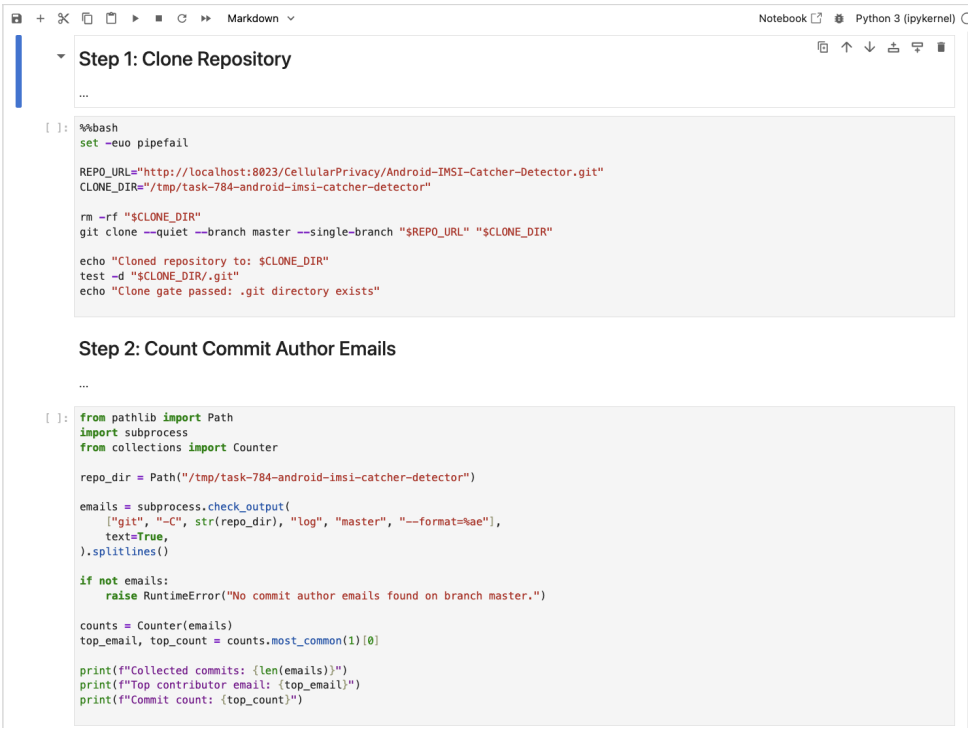


Figure 9: Mixed-language workflow cells in a provisional notebook. For Task 784, the agent can use a Bash cell to clone the GitLab repository `CellularPrivacy/Android-IMSI-Catcher-Detector`, then use Python cells to analyze commits on branch `master` and identify the contributor email. This lets each step use the most suitable tool while preserving the full workflow trace in one artifact.

A.3 SKILL.NB AS AN EXTENSION OF `SKILL.MD`

Agent Skills define a lightweight convention for packaging agent capabilities as folders with a `SKILL.md` file, optional scripts, references, and assets.[2] A SKILL.nb artifact can be viewed as a natural extension of this convention rather than a separate abstraction. It preserves the same core ingredients: a human-readable description, procedural instructions, optional executable resources, and supporting artifacts. The key difference is the execution contract. A `SKILL.md` file may include code blocks, but the markdown artifact alone does not guarantee that an agent executes those blocks exactly as written. Unless an external client or harness enforces verbatim execution, the agent may adapt the snippet, use it as guidance, or ignore it.

SKILL.nb makes this boundary explicit by placing instructions, executable cells, gates, observed outputs, and local evidence in one versioned notebook object. This does not remove the need for task-level execution controls, but it gives maintenance a single auditable artifact for checking what code was intended to run, where validation occurred, and what evidence was produced. The choice also avoids introducing a bespoke trace format: notebooks are a standard and widely supported representation for interleaving markdown, code, metadata, and outputs, and are therefore a familiar substrate for existing tools and LLM-backed agents. Appendix A.4.2 describes the shared skill-loading interface and selective access to notebook cells.

Figure 10 shows the corresponding `SKILL.md` representation for the released Step 1 notebook artifact in Figure 5(b), while Table 4 summarizes the design differences.

The comparison is representational rather than a claim that notebooks are intrinsically safer. Execution still depends on the surrounding client and task environment. The benefit of SKILL.nb is that the

[2] `https://agentskills.io/home`

executable cells, gates, outputs, screenshots, and logs are stored in one artifact, so offline maintenance can audit the intended realization and the evidence used to validate it.

Table 4: Comparison between conventional `SKILL.md` artifacts and SKILL.nb notebook artifacts.

Dimension	**`SKILL.md`**	**SKILL.nb**
Discovery	Name and description are read from markdown front matter or leading prose.	Name, intent, and description are read from the leading notebook markdown cell.
Progressive disclosure	Full-file activation loads all instructions; a chunked or indexed client can select content.	Later cells are loaded only when the workflow, code, gates, or logs are needed.
Procedural guidance	Natural-language instructions are primary.	Natural-language cells remain primary and are interleaved with executable cells.
Execution contract	Code blocks are instructions unless an external client enforces verbatim execution.	Executable cells, gates, and outputs are explicit artifact units available for audit and maintenance.
Execution evidence	External logs or traces must be attached separately.	Cell outputs, screenshots, and validation logs can live in the artifact.
Debugging	Inspecting failures may require separate traces, terminals, or logging systems.	The agent can inspect retained cell outputs, screenshots, and validation logs in the artifact; event capture uses the surrounding instrumentation.
Implementation path	Native format for many skills-compatible clients.	Implemented as an overload of the same skill-loading path, with notebook parsing at activation.
Overhead	Compact plain markdown.	JSON plus cell metadata and outputs, trading size for richer execution structure.

The notebook representation has storage overhead: JSON serialization includes cell metadata and outputs, and raw diffs may be less compact than those of a short `SKILL.md` file. Stored size, however, differs from the text supplied to the model. The loader parses serialization outside model context and returns selected cell contents (Appendix A.4.2). Appendix C.5 compares these quantities for one workflow under full-file Markdown activation and selected-cell notebook access. Selective access depends on the loading interface; a chunked or indexed Markdown client could provide similar selectivity. The notebook keeps procedure, executable code, and retained evidence addressable in one versioned object, while event capture still relies on the surrounding instrumentation.

````
---
name: skillnb-gitlab-merge-request
description: Use this skill to create a GitLab merge request for a known project, source branch, target branch, and reviewer.
metadata:
  site: gitlab
  topic: merge_requests
---

# Create GitLab Merge Request

Use this skill to create a GitLab merge request for a known project, source branch, target branch, and reviewer.

## Inputs

Required inputs:

- `start_url`: initial GitLab URL supplied by the runtime or user, for example `http://localhost:8023`.
- `project_path`: GitLab namespace/project path, for example `a11yproject/a11yproject.com`.
- `source_branch`: source branch to merge from.
- `target_branch`: target branch to merge into.
- `reviewer`: reviewer display name or username to request review from.

## Setup

Run the following to read the inputs from the virtual environment vars and setup the browser page

## Steps

### Step 1. Open Merge Request Setup

#### Intent

```text
Open the GitLab new merge request branch comparison page for the requested project.
```
#### Procedure

1. Derive the GitLab origin from the provided start URL.
2. Navigate to the new merge request page for the requested project path.
3. Wait for the branch comparison controls to load.

#### Metadata

| Key | Value |
| --- | --- |
| `interaction_type` | `navigation` |
| `ui_pattern` | `direct page navigation` |

#### Pre-check

```python
# ... Imports, Playwright automation code are omitted for readability.
print(f"[pre-check] Starting pre-check. Current URL: {page.url}")
if INPUTS.project_path.count("/") < 1:
    raise AssertionError(f"Pre-check failed project_path must include namespace and project, got {INPUTS.project_path!r}.")
# ... Input verification
if await page.locator("a[href*='sign_in'], a[href*='/users/sign_in']").count() != 0:
    raise AssertionError("Pre-check failed GitLab sign-in link is still visible, so the session appears unauthenticated.")
# ... Page state Verification
print("Pre-check passed: authenticated GitLab session is active and all required workflow inputs are present.")
```
````

Figure 10: Markdown skill representation of Figure 5(b), Cells 2–8: released Step 1 with parameterized setup and gates. The `SKILL.md` version stores the same GitLab merge-request workflow as front matter, inputs, natural-language intent and procedure, metadata, and code snippets in a markdown document, whereas the corresponding SKILL.nb artifact stores these units as executable notebook cells with validation gates and cell-local outputs.

A.4 Repository Organization, Search, and Maintenance

This section brings together the persistent workflow artifacts, execution evidence, discovery interface, and maintenance process. The workflow repository $\mathcal{K}$ and execution-log repository $\mathcal{R}_{\text{logs}}$ have different roles: the former holds reusable procedures and their versions, while the latter records what happened during executions. Temporary runtime memory $\mathcal{M}$ is a third, per-run structure. These are distinctions in role and authority; notebook cells can retain or link to execution evidence without making that evidence a released workflow.

A.4.1 WORKFLOW ARTIFACTS AND EXECUTION LOGS

The workflow repository stores the versioned artifacts defined in §3.1: workflow intent I, input schema X, ordered steps S, workflow metadata M^W, and each step's intent, procedure, optional code, gates, and metadata M_i^S. Version links preserve the workflow lineage for review and rollback. Released versions are available for authoritative reuse; provisional artifacts are under validation, and retired artifacts are removed from active retrieval while retained for rollback and analysis. Provisional steps may initially lack gates and acquire them during maintenance before release. Supporting assets become durable through the same maintenance review boundary.

The execution-log repository stores observed executions rather than reusable prescriptions: browser and agent events, action-local outputs, screenshots, page state, gate outcomes, and failure or repair traces. Logs can support diagnosis, workflow creation, step formalization, and maintenance validation, but their presence does not authorize an artifact update. Completed executions also supply the chronological method-native cases used for threshold estimation; benchmark evaluator labels are excluded (Appendix A.5). The online accumulation protocol retains these logs across tasks alongside workflow and lifecycle-policy state; the evaluation protocols specify when persistent state is reset or restored (Appendix B.1).

Runtime memory $\mathcal{M}$ holds transient observations, local repairs, and provisional routines for the current run. It is mutable and non-authoritative, and cannot write directly to $\mathcal{K}$. In existing-workflow mode, runtime executes the latest released version. When no applicable released workflow exists, the agent records a provisional workflow while performing the task and assessing progress from observed environment changes. Similar released workflows, supporting artifacts, and same-site or same-domain logs may supply advisory context. Missing gates are constructed afterward from the recorded execution evidence (Appendix A.5.1). Thus retrieving a trace or a temporary routine does not turn it into a released procedure.

A.4.2 VECTORLESS SEARCH AND AGENT ACCESS

We use vectorless search, implemented with PageIndex (Zhang et al., 2025), over annotated workflows. The index exposes descriptions and navigation evidence that an agent can read and reason about. In the evaluated integration, it combines trace-derived site, navigation, and mutating-action signals with maintenance annotations such as interaction type and UI pattern. Candidate ranking uses navigation-path overlap, and maintenance reasons over the retrieved candidates and their evidence to select relevant prior executions. This retrieval path uses no embedding-distance metric or cutoff. Workflow discovery identifies reusable artifacts in $\mathcal{K}$; lifecycle evidence retrieval selects the related trace set $\mathcal{C}(q)$ from $\mathcal{R}_{\text{logs}}$. The latter is a set of prior executions, not a set of released workflows or the output of embedding-based clustering.

The indexed descriptions draw on workflow and step metadata maintained with the artifact. Maintenance supplies these annotations when it distills and reviews workflows, while recorded executions supply the observed navigation and action signals. Metadata may change when an artifact is updated, repaired, or re-reviewed; Appendix A.7.1 describes its versioning and the separate rules for using it in threshold calibration. This separates describing an observed navigation path for search from validating the workflow that may execute it.

The discovery interface also supports selective access to notebook content. Our implementation overloads the existing skill-loading path so that a notebook can be exposed through the same interface as a `SKILL.md` skill. The agent calls a tool that invokes the notebook CLI and returns the requested cell contents. The CLI uses Jupyter's `nbformat` parser to read notebook serialization outside model context; a `SKILL.md`-only client registers this CLI as its activation tool. At discovery time, the loader extracts the name, intent, and description from the leading Markdown cell; at activation time, it reads the relevant cells and attached resources. Code, gates, outputs, and screenshots therefore remain addressable without loading every stored notebook or its full execution history into the model context. In the merge-request example (Appendix A.1), the reusable header describes the procedure and inputs, while step cells provide the code, checks, and evidence needed for a selected part of the workflow.

A.4.3 Aligning Execution Evidence to Workflow Steps

Once related traces have been selected, maintenance must determine which recorded segments support each workflow step. Playwright browser events and OpenTelemetry agent events share correlation identifiers linking activities to browser events and observed state. Maintenance aligns the resulting bundles to workflow steps rather than matching by embedding distance or sequence position. These identifiers associate observations with activities; they do not by themselves establish that a bundle supports the intended workflow step. Validated segments supporting s_i contribute to n_i^{evidence}, and matched boundary states support gate construction (Appendix A.5.1). We report no annotated alignment-accuracy or cluster-purity measurement. Task success and gate passage therefore do not establish the semantic accuracy of retrieval or alignment.

A.4.4 Experience Distillation

Runtime traces are distilled only into non-authoritative evidence in $\mathcal{M}$ or into candidate workflow-level proposals for offline review. These outputs may summarize reusable procedural fragments, local repairs, missing preconditions, guardrails, or drift signatures, but raw experiences never enter $\mathcal{K}$ directly; durable updates require offline verification and promotion.

A.4.5 Asynchronous Maintenance

Once runtime agents submit a candidate $\mathcal{W}'$, offline maintenance agents verify, refactor, and promote validated workflows into $\mathcal{K}$, with their inference calls counted in maintenance cost. They also review the lessons and repair-derived workflow changes distilled from the run's temporary memory for possible promotion into supporting repository artifacts.

Raw runtime experience is first distilled into non-authoritative per-run evidence in $\mathcal{M}$, and only then do candidate workflow changes or supporting artifacts enter gated promotion. This progression lets SKILL.nb preserve the adaptability of learned agent experience while keeping durable notebook updates and reusable assets under code-level governance.

Verification and Promotion. Upon receiving a proposal $\mathcal{W}'$ or failure-sourced proposal Π_f, maintenance agents refine the candidate against execution traces, stripping exploratory detours and backtracking actions to yield a minimal, reproducible artifact. They resolve duplicates and conflicts against $\mathcal{K}$ and construct missing gates from the recorded evidence before validating the candidate using logged traces, cached regression checks, or side-effect-controlled reruns where available. In the web implementation, these controlled reruns use mocks that mimic the relevant target-system behavior, so a potentially committed action is not repeated in the same live state. Passing a mock-based check supports validation for the behavior represented by that mock; it does not establish equivalence to every live deployment. This consolidation is deliberately stricter than append-only memory: candidate changes must correspond to concrete workflow edits and pass deterministic gate checks before promotion. Gate definitions Γ_i are updated where needed to reflect observed drift. Each promotion creates a new version and release in $\mathcal{K}$ and triggers downstream maintenance jobs.

Threshold crossings initiate review rather than direct writes: release and formalization use trace-support evidence, while demotion and retirement use accepted repair evidence (§3.2). Promotion removes exploratory detours and resolves duplicates or conflicts; it is not a general log-pruning or archival policy.

A.4.6 Artifact Safety and Reproducibility Scope

The workflow artifact schema records intents, procedures, executable cells, gates, metadata, and version links. Hidden evaluator labels and benchmark final answers are excluded from durable workflow fields. Runtime traces and local repair memories are non-authoritative until offline maintenance promotes a validated workflow version into $\mathcal{K}$.

In our evaluation, executable notebook cells are run under the surrounding task or benchmark execution controls. Sandboxing, dependency pinning, credential scoping, and trace redaction require controls in the surrounding deployment harness; the lifecycle policy does not itself provide these controls. The method claims here rely on versioned artifacts, deterministic gate checks, offline validation, and rollback-addressable repository state.

A.5 GATE CONSTRUCTION AND VALIDATION

Table 5 summarizes the signals used by SKILL.nb. Runtime gates are deterministic predicates over browser-observable state and recorded browser events, with expected values grounded in trace evidence or instantiated from task inputs. They can inspect DOM structure, visibility, URLs, form values, page text, counts of relevant page objects, and recorded network requests and response status. Gates do not call benchmark evaluators, hidden success labels, or final-answer oracles.

Maintenance uses logged traces, internal validation, cached regression checks, and controlled reruns. No evaluator-labeled calibration set or extra exploratory round is used in these experiments. Historical cases arise from completed executions in the shared online stream and can inform later decisions; the isolation boundary is between method-native evidence and evaluator data. Hidden task-success labels and final-answer oracles are never inputs to threshold estimation, group assignment, repair acceptance, or promotion. The benchmark evaluator scores outcomes separately.

A repair is accepted only when it passes the triggering trace's deterministic pre/post gates and offline maintenance validation for the affected workflow version. Accepted repairs update the workflow-version repair count used for demotion and the workflow-lineage repair burden used for retirement. A threshold violation is an offline calibration outcome where a candidate threshold would admit a lifecycle action whose downstream validation loss exceeds $J_{\text{perf}}^{\text{ref}} + \epsilon$.

Table 5: Validation and evidence signals used by SKILL.nb. Runtime never accesses benchmark labels. Benchmark labels are metrics-only; threshold cases use chronological method-native evidence.

Signal	**Source**	**Runtime access**	**Use**	**Notes**
Gate pass/fail	Browser state/events + task data	Yes	Runtime validation	Deterministic DOM, URL, visibility, text, form-value, count, and network-event checks
Related trace set $\mathcal{C}(q)$	Logs + retrieval metadata	No	Create	Vectorless search and maintenance selection (Appendix A.4.2)
Step evidence n_i^{evidence}	Distillation + validation	No	Formalize	Counts traces supporting step s_i in the workflow version under consideration
Repair count m_i^{repair}	Accepted repairs	No	Demote	Failed fallbacks do not count; repair must pass maintenance validation
Repair burden ρ_{repair}	Accepted repair token costs	No	Retire	Workflow-lineage aggregate normalized by calibration repair cost
Threshold violation	Offline validation	No	Feasibility	Candidate threshold admits an action exceeding $J_{\text{perf}}^{\text{ref}} + \epsilon$
Benchmark success	Evaluator	No	Metrics only	Final labels are not available to runtime or threshold estimation

A.5.1 CONSTRUCTION FROM RECORDED BOUNDARIES

The step interface is the gate pair $\Gamma_i = (\gamma_{i,\mathrm{pre}}, \gamma_{i,\mathrm{post}})$ over the shared environment, rather than a separate typed input/output signature for each step. The pair belongs to s_i and is shared by its executable realization C_i, NL procedure P_i, and intent I_i. Changing the realization does not change the gate definitions; gate revisions require offline maintenance and promotion of a workflow version. This gives each realization the same observable entry and completion criteria, without proving that the realizations have identical effects.

When no applicable workflow or established gates are available, the agent performs the task without pre-existing workflow gates and records a provisional flow. During this execution, it observes how its actions change the environment and uses these observations to assess progress. Playwright records browser actions, page observations, and network events, while OpenTelemetry records the corresponding agent activity. Available correlation identifiers and timing associate these records with interaction steps.

After this initial execution, an assertion-selection agent in maintenance examines the telemetry and constructs missing gates by selecting a small set of recorded properties that distinguish the intended entry state or outcome of each step. These may include a destination URL, visible page content, a changed field value, or a network request associated with form submission. The agent selects evidence from the recorded execution; it does not generate substitute traces, fabricate observed outputs, or invent values absent from the records.

The backend then checks the selected evidence programmatically against the underlying trace records. For example, if the agent selects a POST request as evidence of form submission, the backend verifies that the referenced request exists and that its selected endpoint and response status match the recorded event. The selection must distinguish the submission request from unrelated network traffic and can include the resulting UI state. Missing or ambiguous evidence is flagged rather than filled in with generated values.

This separates agent-guided assertion selection from programmatic evidence verification and deterministic gate evaluation. Evidence verification establishes that the selected observations were recorded; whether they sufficiently characterize the intended outcome is assessed through artifact validation and the gate-incident analysis in Appendix C.3.

The resulting checks participate in offline validation of the workflow version before promotion and govern subsequent workflow execution. Gates constructed from the initial trace do not retroactively govern the actions that produced that trace. An accepted update becomes part of the versioned artifact; runtime observations alone do not rewrite the repository's gates. Pre- and post-gates are separate executable notebook checks. In the assertion form illustrated below, a failed assertion rejects the boundary state, and completion without an assertion failure accepts it. Gate evaluation makes no model call and uses no hidden evaluator label.

Assertion-selection prompt. The following illustrative prompt preserves the assertion-selection instructions while abstracting deployment-specific details, including where telemetry is retrieved and where selections are written. The exact implementation prompt, including these details, is included in our code repository.

> Analyze the Playwright and OpenTelemetry traces for a task performed in a web environment. For each interaction step, select clear assertions that verify the expected state or outcome using evidence recorded in the traces.
>
> - **Starting page:** Assert the URL matches the main website.
> - **Navigation:** Assert the destination URL and/or visible page content.
> - **Interaction:** Assert the relevant UI change, such as an opened dialog, selected option, or updated field.
> - **Form submission:** Identify the exact POST request representing the submission, distinguish it from unrelated requests, and select assertions for its endpoint, response status, and resulting UI state.
>
> Extract URLs, text, selectors, request details, and expected values exactly from the traces; do not invent them. Prefer the smallest set of reliable assertions per step. Correlate Playwright actions with OpenTelemetry spans using available identifiers and timing. If evidence is missing or ambiguous, flag it.
>
> Return a table: **Step | Action | Selected assertion(s) | Trace evidence/reference**.

Observed failure modes and scope. Determinism makes the predicate inspectable, not semantically complete. Our task-level audit (Appendix C.3) includes two temporal failures: an empty-menu pre-gate passed before a dropdown finished loading, admitting an invalid state; a post-gate requiring "Issue was successfully created" rejected a correctly created issue after that transient message disappeared. These examples motivate checks that distinguish loading from readiness and durable outcomes from transient notifications. A false positive can admit an invalid execution, while a false negative can trigger unnecessary fallback or repair. Such evidence can affect lifecycle review, but gate outcomes do not by themselves authorize a repository update.

A.5.2 Worked Example: Finding and Assigning an Issue

Consider a two-step GitLab workflow: open the requested issue, then assign it to the requested person. For illustration, let X specify a GitLab instance, the issue title "Fix login redirect," and the assignee "Alex Chen." These are illustrative task values, not an additional benchmark run. An aligned trace can show the instance URL before navigation, the issue heading after navigation, the assignee control before assignment, and the displayed assignee afterward. Maintenance uses these observed properties to construct entry and completion checks; at reuse time, expected values come from the current X.

The following schematic cells make the comparisons explicit. The helper functions read, respectively, the current URL, issue-heading text, number of matching assignee controls, and displayed assignee from browser state. They abstract site-specific selectors and are not literal Playwright API calls. The realization between each pair may be code, an NL procedure, or the step intent.

```
# Step 1: open the requested issue
# pre-gate
assert current_url().startswith(X.gitlab_url)
# ... execute the find-issue realization ...
# post-gate
assert issue_heading() == X.issue_title

# Step 2: assign the requested person
# pre-gate
assert assignee_control_count() > 0
# ... execute the assignment realization ...
# post-gate
assert displayed_assignee() == X.assignee
```

For example, a recorded heading of "Fix login redirect" motivates a heading check; on reuse, the check compares the live heading with X.`issue_title` rather than accepting any issue page. Similarly, the existence of an assignee control is an entry condition, while matching the displayed assignee to X.`assignee` is a completion condition. A visible control with the wrong selected person therefore passes this pre-gate but fails the post-gate. These simple checks do not establish unique issue identity or persistence of the assignment; those properties require additional observable conditions when they are part of the intended contract.

A.6 Runtime System Details

This section gives the execution algorithms, gate-conditioned execution semantics, and drift-recovery criterion supporting §3.4. Repository access, experience distillation, and asynchronous maintenance are consolidated in Appendix A.4.

A.6.1 Execution Algorithms

Execution of released workflows with established gates is decomposed into two procedures. Algorithm 1 manages run state, proposal accumulation, and recovery escalation, while Algorithm 2 rechecks gates before each realization and distinguishes ordinary failure from terminal failure after an observed potentially mutating attempt. The top-level loop handles the terminal result before considering global recovery. Failure distillation and the recovery trigger are described in prose rather than as separate algorithm floats.

Algorithm 1 realizes the existing-workflow branch of the runtime loop in §3.4. It manages run-level state: retrieving the released workflow, maintaining the affected-step set $\mathcal{I}_{\text{aff}}$, accumulating local patches, and deciding when to terminate, recover globally, or submit a workflow-level proposal.

Algorithm 2 checks the same pre-gate on the current state before each available realization, including after a local repair. Its status σ_i is success, ordinary failure eligible for bounded recovery, or terminal failure that stops the run. A completed attempt with a passing post-gate succeeds even if it changed external state; the terminal rule applies to failed attempts, including attempts whose post-gate fails. The other outputs are a local patch p_i, a reusable memory item ρ_i, and a successful-local-correction indicator r_i. That indicator supports recovery escalation; it does not itself increment accepted-repair counters, which require maintenance validation.

For an ordinary step failure, runtime extracts a failure-sourced memory item ρ_f and a workflow-level proposal Π_f from the current execution trace. The memory item may later be consolidated into $\mathcal{M}$, while the proposal is submitted for offline maintenance review. Terminal failures retain their trace for maintenance without another live attempt. Recovery escalation uses the normalized affected-step ratio $\rho_{\text{aff}} = |\mathcal{I}_{\text{aff}}|/|S|$ and triggers global recovery when $\rho_{\text{aff}} \geq \tau_{\text{recover}}$, provided the run has not terminated; the next subsection gives the full criterion.

A.6.2 Gate-Conditioned Execution

Runtime handles a query q in either *existing-workflow* or *no-workflow* mode, with repository access and temporary memory as described in Appendix A.4.1. Initial execution without established gates uses agent observations of environment changes to assess progress and supplies traces for subsequent gate construction (Appendix A.5.1). The gate-conditioned rules below apply once the relevant gates exist. Such gates are executable predicates over browser-observable state and cannot query benchmark evaluators, final task labels, or hidden oracle state.

Observed mutations and failure handling. In the web runtime, recorded events such as POST, PUT, PATCH, DELETE, and form submission mark an attempt as potentially state-changing. If that attempt fails, runtime stops the run without retry, local correction, fallback, or global recovery. The same rule governs action-taking local repairs and execution within global recovery. A local repair that leaves the rechecked pre-gate unsatisfied is a failed repair for this purpose. The event check concerns the failed attempt, not all earlier successful actions in the run. Ordinary failure permits consideration of the bounded recovery path; absence of an observed mutation is not proof that no external effect occurred.

This distinction depends on the available action traces and applies to environments exposing equivalent evidence. Effects absent from those traces are outside the safeguard: it does not provide transactional rollback or exactly-once execution. Pre/post gates establish compatibility only for the observable conditions they encode, not full state equivalence or restoration. Frontend state such as an unsaved draft or a consumed single-use control is treated as drift rather than separately guaranteed recoverable. A false-positive post-gate can also accept an incorrect outcome without invoking failure handling. Repository version rollback does not undo external actions.

Algorithm 1 Top-level existing-workflow execution loop

Require: query q, workflow repository $\mathcal{K}$, runtime memory $\mathcal{M}$, execution-log repository $\mathcal{R}_{\text{logs}}$, policy parameter τ_{recover}, env $\mathcal{E}$
Ensure: result y

$\mathcal{W}_{v_L} \leftarrow \text{Retrieve}(q, \mathcal{K})$; $S \leftarrow \mathcal{W}_{v_L}.S$
$\mathcal{I}_{\text{aff}} \leftarrow \varnothing$; patches $\leftarrow \varnothing$; $\mathcal{B}_{\text{mem}} \leftarrow \varnothing$
for $i = 1$ **to** $|S|$ **do**
  $\mathcal{N}_i \leftarrow \text{RetrieveMem}(q, x_t, S[i], \mathcal{M})$
  $(\sigma_i, p_i, \rho_i, r_i) \leftarrow \textsc{PerformStep}(\mathcal{W}_{v_L}, i, \mathcal{N}_i, \mathcal{E})$
  if $\sigma_i = \text{terminal}$ **then**
    retain the failure trace for offline maintenance; **return** failure
  end if
  if $\sigma_i = \text{failure}$ **then**
    distill failure memory ρ_f and proposal Π_f from $(q, i, x_t, \mathcal{R}_{\text{logs}})$
    add non-empty ρ_f to $\mathcal{B}_{\text{mem}}$; submit non-empty Π_f
    $\mathcal{I}_{\text{aff}} \leftarrow \mathcal{I}_{\text{aff}} \cup \{i\}$
  else
    add non-empty ρ_i to $\mathcal{B}_{\text{mem}}$ and non-empty p_i to patches
    if r_i **or** $p_i \neq \varnothing$ **then**
      $\mathcal{I}_{\text{aff}} \leftarrow \mathcal{I}_{\text{aff}} \cup \{i\}$
    end if
  end if
  $\rho_{\text{aff}} \leftarrow |\mathcal{I}_{\text{aff}}|/|S|$
  if $\rho_{\text{aff}} \geq \tau_{\text{recover}}$ **then**
    $(p_g, y_g) \leftarrow \text{GlobalRecoverAndExecute}(\mathcal{W}_{v_L}, i, q, \mathcal{N}_i, \mathcal{R}_{\text{logs}}, \mathcal{E}, \mathcal{K})$
    if $p_g \neq \varnothing$ **then**
      patches $\leftarrow$ patches $\cup \{p_g\}$
    end if
    if $y_g \neq \text{success}$ **then**
      if $\mathcal{B}_{\text{mem}} \neq \varnothing$ **then**
        $\mathcal{M} \leftarrow \text{Consolidate}(\mathcal{M}, \mathcal{B}_{\text{mem}})$
      end if
      return failure
    end if
    break
  else if $\sigma_i = \text{failure}$ **then**
    if $\mathcal{B}_{\text{mem}} \neq \varnothing$ **then**
      $\mathcal{M} \leftarrow \text{Consolidate}(\mathcal{M}, \mathcal{B}_{\text{mem}})$
    end if
    return failure
  end if
end for
if patches $\neq \varnothing$ **then**
  $\mathcal{W}' \leftarrow \text{BuildProposal}(\mathcal{W}_{v_L}, \text{patches}, \mathcal{I}_{\text{aff}})$
  $\text{Submit}(\mathcal{W}')$
end if
if $\mathcal{B}_{\text{mem}} \neq \varnothing$ **then**
  $\mathcal{M} \leftarrow \text{Consolidate}(\mathcal{M}, \mathcal{B}_{\text{mem}})$
end if
return success

Algorithm 2 Existing-workflow step execution with per-attempt gates and terminal failure handling

Require: released workflow $\mathcal{W}_{v_L}$ with established gates, step index i, retrieved memory $\mathcal{N}_i$, env $\mathcal{E}$
Ensure: $(\sigma_i, p_i, \rho_i, r_i)$, where $\sigma_i \in \{\text{success}, \text{failure}, \text{terminal}\}$

$p_i \leftarrow \varnothing$; $\rho_i \leftarrow \varnothing$; $r_i \leftarrow \text{false}$
for available realization a in $C_i \to P_i \to I_i$, within the retry budget **do**
  observe current state x_t
  if not $\gamma_{i,\text{pre}}(x_t)$ **then**
    diagnose mismatch; attempt local repair only for eligible environmental drift
    record repair events and any induced patch p_i; refresh x_t
    if no repair succeeds or $\gamma_{i,\text{pre}}(x_t)$ still fails **then**
      if the failed repair recorded a potentially mutating event **then**
        return $(\text{terminal}, p_i, \rho_i, r_i)$
      end if
      return $(\text{failure}, p_i, \rho_i, r_i)$
    end if
    $r_i \leftarrow \text{true}$
  end if
  execute a and record its action events; refresh x_t and evaluate $\gamma_{i,\text{post}}(x_t)$
  if the attempt completes and $\gamma_{i,\text{post}}(x_t)$ passes **then**
    distill any reusable local-repair memory into ρ_i
    return $(\text{success}, p_i, \rho_i, r_i)$
  end if
  if the failed attempt recorded a potentially mutating event **then**
    return $(\text{terminal}, p_i, \rho_i, r_i)$
  end if
end for
return $(\text{failure}, p_i, \rho_i, r_i)$

A.6.3 Runtime Recovery Criterion

During execution, runtime maintains a run-level instability signal to determine when local repair is no longer sufficient. A step is marked as affected when it fails locally, requires a successful local correction, or produces a non-empty step-local patch p_i indicating drift. Let

$$\mathcal{I}_{\text{aff}} = \{i \mid \text{step } i \text{ has shown instability in the current run}\}$$

denote the set of unique affected steps in the current run. Each step contributes at most once, so repeated local difficulty on the same step does not artificially inflate the run-level signal. The normalized affected-step ratio is

$$\rho_{\text{aff}} = \frac{|\mathcal{I}_{\text{aff}}|}{|S|},$$

where $|S|$ is the number of steps in the current workflow.

Unless the run has terminated, runtime escalates to workflow-level recovery when this ratio crosses the runtime recovery control τ_{recover}:

$$e_i = (\rho_{\text{aff}} \geq \tau_{\text{recover}}).$$

This normalization makes the recovery control comparable across workflows of different lengths. When e_i holds, the agent invokes a global recovery routine that re-plans and executes the remaining steps using retrieved memories, execution history, and prior workflow versions as context (Ouyang et al., 2025). If recovery cannot make progress under the available task context and permissions, the run terminates as a failure. When progress requires an external precondition unavailable to the agent, such as credentials or user approval, runtime pauses for operator input or terminates. This fallback is outside the autonomous lifecycle policy and is not counted as a method capability in evaluation.

A.7 Adaptive Thresholds: Technical Details

This section records the deployed threshold-estimation procedure supporting §3.3. Thresholds are filtered by replay on logged decision opportunities. The deployment rule then uses a pooled default, blends feasible group and pooled estimates, or defers to maintenance review according to the available support and feasible sets.

Shrinkage and projection. After the pooled and group sweeps select feasible thresholds, the specialized branch combines them using the sample-size weight $\omega_{g,d} = n_{g,d}/(n_{g,d} + n_0)$. This is a partial-pooling rule (Gelman et al., 2013) with reference weight n_0 specified directly. For $n_{g,d} > 0$ and $n_0 > 0$, its blend $\tilde{\tau}_d^g$ has the exact algebraic characterization

$$\begin{aligned}\tilde{\tau}_d^g &= \arg\min_{t\in\mathbb{R}} \left\{ n_{g,d}(t - \hat{\tau}_d^g)^2 + n_0(t - \hat{\tau}_d^{\text{pool}})^2 \right\} \\ &= \frac{n_{g,d}}{n_{g,d} + n_0}\hat{\tau}_d^g + \frac{n_0}{n_{g,d} + n_0}\hat{\tau}_d^{\text{pool}}.\end{aligned} \tag{1}$$

Differentiating the quadratic gives $n_{g,d}(t-\hat{\tau}_d^g)+n_0(t-\hat{\tau}_d^{\text{pool}}) = 0$, which yields the stated blend. The quadratic also equals $(n_{g,d}+n_0)(t-\tilde{\tau}_d^g)^2$ plus a constant independent of t. Consequently, projecting $\tilde{\tau}_d^g$ onto the nearest value in $\mathcal{F}_{g,d}$ minimizes this same quadratic over the feasible candidates. This characterizes the blending stage after the maintenance-cost sweeps have selected the two estimates; the sweeps and the blend serve distinct roles. The feasibility branches and tie-breaking rule are given below and in Table 6.

A.7.1 Threshold Estimation Procedure

Before threshold estimation, offline maintenance agents assign each workflow and step to a group using the versioned metadata maps stored with the artifact. These group descriptors are maintenance-produced artifact metadata, not a fixed benchmark taxonomy. They may change when a workflow version is updated, repaired, or re-reviewed. Descriptors are selected from a constrained schema and canonicalized before threshold estimation; the schema and canonicalization rules are frozen for each calibration run. For threshold calibration, group assignment is performed chronologically from metadata available at the time of the logged decision. Later outcomes, benchmark labels, and final task success are excluded from the grouping input. Rare or novel descriptors back off to coarser parents, and the exact group assignment used for each decision opportunity is logged with the workflow version. Descriptors are restricted to reusable, observable properties such as site family, task type, action type, step type, and interface properties such as whether a form is dynamic. They do not include benchmark task identifiers, hidden evaluator outputs, final answers, or success labels. The resulting labels determine which logged decision opportunities contribute to group-specific estimates.

Decision records and support. For each group g and lifecycle decision d, estimation uses completed cases available before the current decision. Each usable case contains the signal u_j, candidate action, attributable maintenance-token cost, and offline validation outcome, with its group assignment recorded against the workflow version. A case missing a required field is excluded from $\mathcal{D}_{g,d}$. The calibration support $n_{g,d} = |\mathcal{D}_{g,d}|$ counts decision cases, whereas u_j is the evidence signal within a case. For example, one creation case supported by eight related traces has $u_j = 8$ and contributes one case to $n_{g,d}$, not eight. This separates the evidence needed to trigger an action from the sample size used to calibrate its threshold.

Maintenance cost and validation outcomes. At the event level, the cost proxy is $c(e) = \text{tok}_{\text{in}}(e) + \text{tok}_{\text{out}}(e)$, including inference performed by maintenance agents. For each candidate threshold, replay estimates $\hat{C}^{(d)}_{\text{maint}}(g, \tau)$ from the attributable maintenance costs, omitting costs common to all candidates. This cost ranks thresholds after the validation filter is applied; it does not include an estimate of how suppressed actions would change future runtime trajectories.

The validation outcome determines whether a logged candidate exceeds the performance tolerance. Writing $J_{\text{perf},j}$ for the observed offline validation loss in case j, define

$$b_j = \mathbf{1}\left[J_{\text{perf},j} > J^{\text{ref}}_{\text{perf}} + \epsilon\right], \qquad a_j(\tau) = \mathbf{1}[u_j \geq \tau], \qquad k_{g,d}(\tau) = \sum_{j\in\mathcal{D}_{g,d}} a_j(\tau) b_j.$$

Here ϵ sets the tolerated loss increase for an individual validation comparison. In contrast, $V^{(d)}_{\max}$ limits the Wilson upper bound on violation incidence across usable cases, and α sets that bound's tail probability. These parameters have distinct roles. Validation uses the method-native evidence described in Appendix A.5, not hidden benchmark labels.

Candidate sweep. The retirement normalizer is fixed from calibration logs as $c_{\text{ref}} = \max_{e \in \mathcal{R}_{\text{cal}}} c(e)$, the maximum accepted repair-event token cost observed during calibration. If $\mathcal{R}_{\text{cal}}$ is empty, automatic retirement by repair burden is deferred to maintenance review. For each decision d, the candidate set $\mathcal{T}_d$ is the sorted set of unique replay values of that decision's signal, not a hand-tuned grid. Each candidate is evaluated on the same usable set $\mathcal{D}_{g,d}$: changing τ changes the admitted cases and $k_{g,d}(\tau)$, but not $n_{g,d}$. The sweep keeps candidates whose pointwise one-sided Wilson upper bound is at most $V_{\max}^{(d)}$ and selects the feasible candidate with lowest replay-estimated maintenance cost. The same sweep is run on pooled data and on each group with usable support.

Violation-rate denominator. For a candidate threshold τ, let $n_{g,d}^{\text{adm}}(\tau)$ count usable cases with $u_j \geq \tau$. The denominator of $\hat{V}^{(d)}(g, \tau) = k_{g,d}(\tau)/n_{g,d}$ includes all usable logged cases, including those suppressed by τ; only admitted cases exceeding the validation tolerance contribute to the numerator. Thus, $\hat{V}^{(d)}$ measures violation incidence per logged opportunity. The fraction of admitted cases that violate the tolerance instead uses $k_{g,d}(\tau)/n_{g,d}^{\text{adm}}(\tau)$, defined when $n_{g,d}^{\text{adm}}(\tau) > 0$. A fully observed case suppressed by a candidate threshold remains in $n_{g,d}$; a case excluded for missing cost or validation information does not.

For $k = k_{g,d}(\tau)$, $n = n_{g,d}$, $\hat{p} = k/n$, and $z = \Phi^{-1}(1-\alpha)$, the one-sided Wilson upper bound is

$$\text{WilsonUCB}_{1-\alpha}(k, n) = \frac{\hat{p} + z^2/(2n) + z\sqrt{\hat{p}(1-\hat{p})/n + z^2/(4n^2)}}{1 + z^2/n}.$$

A threshold is feasible when this upper bound is at most $V_{\max}^{(d)}$. Because the final threshold is selected after sweeping over $\mathcal{T}_d$, this bound is used as a conservative pointwise filter rather than a formal uniform confidence guarantee after adaptive selection.

Illustrative feasibility check. Consider 100 usable cases, of which 20 cross a candidate threshold and two exceed the validation tolerance. The observed violation incidence is $2/100 = 2\%$, while the fraction among admitted cases is $2/20 = 10\%$. With the experimental settings $\alpha = 0.05$ and $V_{\max}^{(d)} = 0.05$, the one-sided Wilson upper bound for $k = 2$ and $n = 100$ is approximately 5.86%, so this threshold is excluded despite its 2% point estimate. A candidate admitting only one violating case among the same 100 usable cases has an upper bound of approximately 4.36% and passes the filter. The sweep compares maintenance costs among the candidates that pass; a lower estimated cost cannot make an infeasible candidate eligible. These counts illustrate the calculation and are not an additional experimental result.

The deployed rule uses the sample-size shrinkage weight $\omega_{g,d} = n_{g,d}/(n_{g,d} + n_0)$ from §3.3. Within the feasible specialized branch, smaller support increases the pooled weight and larger support increases the weight on the group estimate. Equation 1 characterizes this blend in terms of the already-selected thresholds and the specified reference weight n_0. After shrinkage, the blended threshold may not be one of the replay-feasible candidate values, so the deployed rule projects it back onto the finite feasible set:

$$\Pi_{\mathcal{F}}(x) = \arg\min_{\tau \in \mathcal{F}} |\tau - x|.$$

Ties are broken toward $\hat{\tau}_d^{\text{pool}}$. Table 6 summarizes the branches in §3.3. A group with no usable cases can use a feasible pooled default; a group with usable cases but no feasible threshold defers to maintenance review. The value defer_d is not a numeric threshold: no automatic thresholded action is taken for that decision, and the case is routed to maintenance review. Threshold crossings also trigger review; repository changes require validation and promotion.

Table 6: Threshold deployment after replay filtering. A nonempty feasible set contains at least one candidate satisfying the stated Wilson filter.

Available support and feasibility	Deployed behavior
Pooled feasible set is empty	Defer to maintenance review, regardless of group support.
Pooled feasible set is nonempty; $n_{g,d} = 0$	Use the pooled threshold.
Pooled and group feasible sets are nonempty; $n_{g,d} > 0$	Blend group and pooled estimates, then project onto the group-feasible set.
Pooled feasible set is nonempty; $n_{g,d} > 0$ but the group feasible set is empty	Defer to maintenance review.

Maintenance review and evidence updates. Crossing a deployed threshold makes a candidate eligible for maintenance review; it does not accept the candidate or modify the repository. Deferral also routes the case to maintenance review, but without an automatic thresholded action. In either path, repository changes must satisfy the verification and promotion requirements in Appendix A.4.5. Maintenance agents inspect the supporting evidence and validate the proposed artifact change; threshold feasibility does not replace those checks.

Successful executions supply workflow-level evidence for creation and step-level evidence for formalization. Only repairs accepted by maintenance add to the repair-based signals: m_i^{repair} counts accepted repairs for a step in the workflow version under consideration, while ρ_{repair} aggregates their token-weighted burden along the workflow lineage. A failed fallback or an unvalidated local patch does not itself increment these accepted-repair signals. Table 7 connects each signal to its lifecycle action and validation requirement.

Table 7: Lifecycle decisions governed by replay-calibrated thresholds.

Decision	Signal	Threshold	Action	Required validation
Create	$\lvert\mathcal{C}(q)\rvert$	Count	provisional $\rightarrow$ released	ValidW_v
Form	n_i^{evidence}	Count	add/use $C_{i,v}$	$\text{Valid}\text{C}_{i,v}$
Demote	m_i^{repair}	Count	executable $\rightarrow$ NL-guided successor	accepted repairs are gate- and validation-approved
Retire	$\rho_{\text{repair}}(\mathcal{W}_v)$	Burden	released $\rightarrow$ retired	RetireOK_v after maintenance review

A.7.2 Replay-Relative Scope of the Feasibility Check

Projection does not claim the threshold is safe under future shifts. It only ensures that the deployed value is one of the candidates that passed the replay filter on the logged opportunities used for estimation. The UCB check is replay-relative: it filters thresholds using $\hat{V}^{(d)}$, the violation rate estimated by changing one candidate threshold at historical decision points while keeping the rest of the logged repository trajectory fixed. It does not control the true violation rate under arbitrary future workload or interface shift, and it does not model second-order effects where a different threshold would have changed later repository contents, group assignments, or repair opportunities. For this reason, the paper describes the result as UCB-bounded replay feasibility rather than a deployment-time safety guarantee.

A.7.3 Why Threshold Control Rather Than End-to-End RL

Recent RLVR methods update model parameters using verifiable task rewards (Lambert et al., 2025; Guo et al., 2025). Table 8 summarizes why SKILL.nb uses thresholded lifecycle control rather than end-to-end RL or RLVR. The distinction is not that RLVR is unsuitable in general. Rather, the policy target here is a durable repository of workflow artifacts, and the available counterfactual evidence is local to logged lifecycle decisions.

Table 8: Why SKILL.nb uses thresholded lifecycle control rather than end-to-end RLVR. The policy governs durable workflow artifacts, not token generation or primitive browser actions.

Property	End-to-end RL/RLVR	SKILL.nb threshold control
Policy target	Model or browser-action policy	Repository lifecycle policy
Action granularity	Token or primitive action level	Release, formalize, demote, retire
Feedback signal	Task-level verifiable reward	Gates, accepted repairs, and replay validation
Policy search	Online exploration or broad offline optimization	Constrained replay-calibrated threshold selection
Counterfactual burden	Long-horizon trajectory effects	Logged lifecycle decision opportunities
Safety scope	Depends on verifier and exploration regime	Replay-relative UCB filtering plus runtime gates
Interpretability	Learned policy may be opaque	Auditable thresholds by group and decision

A.8 Limitations and Scope

SKILL.nb's lifecycle decisions are bounded by logged execution evidence. The Wilson-UCB feasibility check bounds estimated violation rates on threshold-estimation cases, not arbitrary future workloads or interface shifts. The method also depends on reliable gates, metadata quality, group assignment, and recurring task structure; gate errors can admit invalid executions, while available replay support and feasibility determine pooled, specialized, or deferred lifecycle decisions. Our maintenance-cost proxy counts LLM inference tokens rather than total operational cost, such as wall-clock latency, storage, or human review. Finally, our evaluation studies reuse and lifecycle correction in controlled benchmark environments, including single-application GitLab version migration, which may differ from long-term production deployment.

The adaptive-threshold procedure has bounded scope in several important ways. Its UCB-bounded filtering is replay-relative through the estimator $\hat{V}^{(d)}$, rather than a blanket deployment-time guarantee under arbitrary distribution shift. Replay changes one candidate threshold at a time while holding the rest of the logged lifecycle fixed, so it does not model second-order effects on future repository contents, future group assignments, or later repair opportunities. The offline routine sweeps candidates separately for each decision, then, where applicable, applies the group–pooled blend and feasibility projection described in Appendix A.7. We also do not evaluate alternative threshold estimators such as Bayesian optimization over the empirical replay breakpoints or learned predictors from group features. Direct offline partial pooling with UCB-constrained sweeps is chosen here for sample efficiency on finite replay candidate sets, interpretability, and explicit feasibility filtering. Appendix A.7 provides the technical details.

The method also depends on the quality of the gates and metadata used to produce these signals. Evidence and repair signals are only as reliable as the gate specifications $(\gamma_{i,\mathrm{pre}}, \gamma_{i,\mathrm{post}})$ and the offline metadata tags used to group workflows and steps. Gate false positives can admit an invalid step execution or repair, while false negatives can trigger unnecessary fallback, demotion, or repair. Poor gates or miscalibrated tags can therefore distort the signals driving the lifecycle decisions. Separately, the benefits of specialization depend on recurring workload structure. This assumption is plausible in deployment settings such as help-desk or enterprise automation, where many requests recur as variations of the same underlying workflows, but it may fail in more heterogeneous environments. Groups without usable cases can use a feasible pooled default. Groups with usable cases enter the blend only when both group and pooled feasible sets are nonempty; otherwise, the lifecycle decision defers to maintenance review (Table 6). The maintenance-cost proxy $c(e) = \mathrm{tok}_{\mathrm{in}}(e) + \mathrm{tok}_{\mathrm{out}}(e)$ measures LLM inference tokens only. It does not optimize for wall-clock latency, human handoff cost, storage overhead for the repository and logs, or developer review effort. For retirement, token-weighted repair burden is a maintenance-effort proxy rather than a complete measure of semantic criticality, so a cheap but critical repair can still be underweighted.

The evaluation tests lifecycle correction through repeated execution, bounded repair, the joint demotion/retirement ablation, and controlled GitLab version migration. These controlled experiments do not characterize long-term production repository growth or isolate the individual effects of demotion and retirement.

Broader impacts. More auditable workflow artifacts can make web agents easier to inspect, reproduce, and roll back when they fail. This may reduce silent regressions in benign automation settings such as internal support or software-maintenance workflows. The same capability could also make undesired web automation more reliable, or preserve traces that contain sensitive visual state or credentials if deployed without redaction. For this reason, our claims are limited to controlled benchmarks, and deployment should pair lifecycle governance with credential scoping, trace redaction, rate limits, and task-level authorization controls.

B Experimental Details

This appendix records protocol details for the main experiments in §4. The main text summarizes the benchmark choices, shared harness, and statistical conventions, while this appendix pins the run configuration, persistent-state rules, re-execution protocol, repair protocol, and confidence-interval calculations. Additional component-removal and transfer diagnostics appear in Appendix C.

B.1 Evaluation Protocol

All compared methods use the same GPT 5.3 Codex model, tool access, and task budget in a shared evaluation harness. Reasoning effort, temperature, and update schedules remain method-specific, as recorded below. The main-text tables and figures report this shared-harness re-evaluation rather than numbers copied directly from the original papers. For baseline methods, the shared harness wraps the released public implementations and preserves their native persistent-state and update paths rather than replacing them with a common re-implementation. On WebArena-Verified, the benchmark table in the main text reports a fresh-start full benchmark round in which each method starts without persistent learned state and builds any repository or memory online during that round. Separately, the lifecycle protocol runs for five rounds over the full 812-task benchmark. For persistent-state methods, the GitLab version-drift evaluation includes fresh-start runs on GitLab 15.7, GitLab 16.11, and GitLab 18.9, plus old-to-target conditions that build old-version state on GitLab 15.7 and reuse restored snapshots on GitLab 16.11 and GitLab 18.9. We choose GitLab 16.11 as an intermediate target because documented 16.x interface changes overlap with GitLab task families in WebArena-Verified, and use GitLab 18.9 as a longer-horizon target. Appendix B.4 gives the detailed GitLab protocol and version-selection rationale. The benchmark environment is reset for each task execution. What persists varies by method: $\text{AWM}_{\text{online}}$ carries its induced workflow memory, ReasoningBank carries its distilled reasoning bank, and SKILL.nb carries its workflow repository $\mathcal{K}$, associated workflow state, event logs, and lifecycle-policy state. CodeAct has no persistent learned state or released-artifact maintenance loop and is therefore omitted from the lifecycle figures and GitLab old-state reuse evaluation. We do not synchronize update frequencies across methods. Each persistent-state method updates its state according to its own design during the multi-round protocol.

Task order and online state. In the online and five-round lifecycle evaluations, each WebArena-Verified round uses a shuffled task order, and the same order for that round is applied to all methods. The order is reshuffled between rounds. After each task, transient agent context is cleared; persistent method state is not reset between tasks or rounds. The first-request-only setting instead clears persistent state before every task (Appendix C.4). We do not report multiple independent order seeds in this draft, so the online results should be interpreted as performance under the shared online-learning orders used here rather than as order-invariant estimates. Persistent state is allowed to store reusable task-family structure, natural-language procedures, executable workflow components, gates, and repair history. It is not allowed to store hidden evaluator outputs, benchmark success labels, final answers, or benchmark-specific task identifiers as reusable workflow fields. URLs may appear only when they are part of the observable task state or a reusable site-level navigation pattern.

Lifecycle perturbations. For each lifecycle round, we generate one perturbation set offline and apply it to every method. Eligible tasks receive a same-site start URL sampled by random walk; tasks that require a benchmark-specified non-default start URL keep that required condition. Language perturbations are generated at the WebArena-Verified `intent_template` level, e.g., "Get name(s) of reviewer(s) who mention {{description}} for the product on the current page," rather than by rewriting fully instantiated tasks. An LLM proposes paraphrases while preserving slot variables, intent, and task parameters. We filter candidates with a sentence encoder, require semantic similarity above 90%, and choose paraphrases with a penalty on word-level similarity so the maximum planned

Table 9: Reproducibility record for the reported experiments. For baseline methods marked with *, temperature settings follow the corresponding original work where specified. All methods are re-evaluated with `gpt-5.3-codex`, with method-specific reasoning effort and temperature settings listed below.

Field	Value
Methods	
CodeAct*	
Model	`gpt-5.3-codex`
Reasoning effort	`high`
Temperature	0.6 (temperature not reported in the original paper)
AWM$_{\text{online}}$*	
Model	`gpt-5.3-codex`
Reasoning effort	`high`
Temperature	0.0 for action generation, workflow induction, and memory update
ReasoningBank*	
Model	`gpt-5.3-codex`
Reasoning effort	`high`
Temperature	0.7 for generation, 1.0 for memory extraction, 0.0 for classification
SKILL.nb	
Model	`gpt-5.3-codex`
Reasoning effort	`medium` for execution, `high` for repair and drift handling
Temperature	0.2 for execution, creation, and formalization, 0.6 for repair and drift handling
Lifecycle thresholds	$V^{(d)}_{\max} = 0.05$, $\alpha = 0.05$, $n_0 = 2$, $\tau_{\text{recover}} = 0.25$
Agent implementation	`opencode`[3]
Benchmarks and evaluators	
WebArena-Verified	Commit `6473f72db5dcefc97b5725b59e734504edc28a21`[4]
WebArena-Verified hard subset	258 tasks for component and threshold diagnostics
GitLab single-site subset	180 WebArena-Verified tasks from the full dataset supporting the main controlled version-drift evaluation
Mind2Web	Commit `33bd95caeee7bba22dd08ecc935845e15c5e5dc7`[5]
Execution protocol	
Max steps	30 per task
Browser environment	Playwright MCP-controlled browser[6] with per-task environment reset

number of rounds uses diverse but semantically close wording. The paraphrase bank is generated once before evaluation and then reused across methods.

The exact template-paraphrasing prompt is:

> You are paraphrasing WebArena-Verified intent templates for controlled evaluation. Rewrite the template in different wording while preserving the exact task intent, all required outputs, and every slot variable exactly as written. Do not add constraints, remove constraints, change entities, change the requested output type, or alter the success condition. Keep all {{slot}} placeholders unchanged. Return N paraphrases, one per line.

For SKILL.nb, the experimental hyperparameters are fixed unless otherwise noted: $V^{(d)}_{\max} = 0.05$ for all lifecycle decisions, $\alpha = 0.05$, $n_0 = 2$, and $\tau_{\text{recover}} = 0.25$. These values define the replay-feasibility budget, UCB tail probability, shrinkage reference sample size, and runtime recovery trigger used throughout the experiments. The SKILL.nb runtime and maintenance agents are implemented using `opencode`.

Reproducibility record. Table 9 records the fixed run configuration used for the reported experiments.

For SKILL.nb, the persistent state specifically includes the workflow repository $\mathcal{K}$, workflow states (including provisional, released, and retired artifacts), supporting event logs, and the learned lifecycle-policy state. Reuse consistency is measured before any update. For each method, the denominator is the set of tasks solved in the initial WebArena-Verified round. We freeze the method-specific learned state used for each such success: the released workflow for SKILL.nb, the associated workflow-memory state for $\text{AWM}_{\text{online}}$, and the retrieved reasoning-bank state for ReasoningBank. Each frozen artifact is re-executed three times under independent environment resets, shared start-URL perturbations, and shared intent-template paraphrases. A task is counted as maintaining reuse consistency only if all three re-executions succeed without editing the artifact.

In the repair analysis, each failing artifact receives up to three repair attempts through that method's native state-update mechanism. We do not impose a synthetic cross-method validation gate. For SKILL.nb, a candidate repair must pass promotion before replacing the pinned workflow version. For $\text{AWM}_{\text{online}}$, repair attempts use its workflow-memory update path; for ReasoningBank, they use its memory extraction and reasoning-bank update path.

The reported recovery denominators are 110 for $\text{AWM}_{\text{online}}$, 95 for ReasoningBank, and 96 for SKILL.nb; the corresponding regression denominators are 240, 220, and 260.

Recovery at budget B is the fraction of failed artifacts restored to success within B native update attempts. After each accepted update attempt, regression is evaluated on the cached-trace subset of tasks that were passing before the candidate update. This subset is method-specific: it is drawn from that method's previously passing tasks with reusable cached traces available at repair time and therefore varies by update event. Reuse consistency, recovery, and regression are therefore reported with method-conditional denominators, and the repair/regression comparison characterizes native update behavior rather than a matched validation-gate ablation.

On Mind2Web, we report only a single benchmark round in which each method starts without task-specific persistent state. Methods may accumulate workflows or memories online as allowed by their design, but no re-execution or repair is evaluated there. All repeated-execution and repair results in this paper are therefore specific to WebArena-Verified under the shared evaluation setup.

For statistical reporting on WebArena-Verified, individual method rates use nominal Wilson 95% binomial intervals. These intervals and the task-paired test below do not model dependence induced by online state updates. The main overall success-rate comparison between SKILL.nb and the next-best baseline is additionally evaluated with a two-sided continuity-corrected McNemar test on per-task success outcomes over the shared 812-task benchmark. We do not report analogous paired tests for every per-site subset in this draft, so those breakdowns should be interpreted as descriptive point estimates.

The round-2 dip for SKILL.nb in Figure 1(a) coincides with demotions of workflows created late in round 1. Because several task templates appear only once or twice, those workflows had limited evidence before the next perturbed round. We treat this as a protocol-specific diagnostic rather than a standalone claim about the method.

B.2 Lifecycle Curve Confidence Intervals

Table 10 reports the 95% Wilson confidence intervals for the per-round success rates shown in Figure 1(a). Each cell is computed over the full 812-task WebArena-Verified round for that method and round. The McNemar test reported in the main text is computed from paired per-task outcomes over the same 812 tasks, while the marginal Wilson intervals for the single-round WebArena-Verified benchmark are reported directly in Table 1.

B.3 Recorded Inference Costs

We measure total recorded inference tokens per successful task across the five rounds of the lifecycle protocol, including execution and maintenance inference. Table 12 reports the values under each

[3] https://github.com/opencode-ai/opencode

[4] https://github.com/ServiceNow/webarena-verified

[5] https://github.com/OSU-NLP-Group/Mind2Web

[6] https://github.com/microsoft/playwright-mcp

Table 10: Per-round WebArena-Verified success rates with 95% Wilson confidence intervals for Figure 1(a). Each entry is reported as SR [lower, upper], in percent.

Method	Round 1	Round 2	Round 3	Round 4	Round 5
AWM_{online}	46.4 [43.0, 49.9]	43.8 [40.5, 47.3]	45.1 [41.7, 48.5]	44.0 [40.6, 47.4]	39.4 [36.1, 42.8]
ReasoningBank	49.8 [46.3, 53.2]	49.4 [46.0, 52.8]	48.2 [44.7, 51.6]	45.6 [42.2, 49.0]	45.2 [41.8, 48.6]
SKILL.nb	53.7 [50.3, 57.1]	52.0 [48.5, 55.4]	54.3 [50.9, 57.7]	55.4 [52.0, 58.8]	55.7 [52.2, 59.0]

Table 11: Budget-2 recovery and regression rates with 95% Wilson confidence intervals for Figure 1(c). Entries are reported as rate [lower, upper], in percent. All entries are displayed to one decimal for consistency; baseline point estimates retain whole-percentage rounding.

Method	Recovery at budget 2	Regression at budget 2
AWM_{online}	58.0 [48.8, 67.0]	17.0 [12.8, 22.4]
ReasoningBank	63.0 [53.1, 72.2]	15.0 [10.9, 20.3]
SKILL.nb	72.9 [63.3, 80.8]	4.2 [2.4, 7.4]

method's native configuration. Benchmark evaluation and post hoc analysis are excluded, as are browser/tool action cost, wall-clock latency, storage, human handoff, and developer effort. This measure is distinct from the internal update costs in the component ablation and the maintenance measure in the threshold ablation.

Table 12: Recorded inference tokens per success (thousands) under native configurations. The final column is a success-weighted approximation from rounded per-round costs.

Method	R1	R2	R3	R4	R5	All five rounds
AWM_{online}	42	42	41	42	41	41.6
ReasoningBank	44	43	40	39	40	41.3
SKILL.nb	52	55	41	36	36	43.8

For method m, the five-round estimate is $\bar{c}_m = \sum_{r=1}^{5} s_{m,r} c_{m,r} / \sum_{r=1}^{5} s_{m,r}$, where $s_{m,r}$ is its success count and $c_{m,r}$ its reported tokens per success in round r. This approximates total recorded tokens divided by total successes, rather than averaging the per-round ratios equally. Since the reported costs are rounded, the aggregate is not an exact token measurement.

SKILL.nb's per-success cost rises from 52k to 55k in round 2 and then falls to 36k by round 5, 30.8% below round 1. It has the lowest per-round cost among these methods in rounds 4–5, but its approximately 43.8k five-round aggregate exceeds AWM_{online}'s 41.6k and ReasoningBank's 41.3k. Thus the per-round crossover does not establish a cumulative five-round cost advantage. Different reasoning settings and update schedules also preclude an equal-compute interpretation or a claim about total operational cost.

Figure 11 shows the same SKILL.nb series normalized to its own round-1 value: round 5 reaches 69.2% of round 1. This is a within-method inference-cost trend, not maintenance-only accounting or evidence that repository stabilization caused the decline.

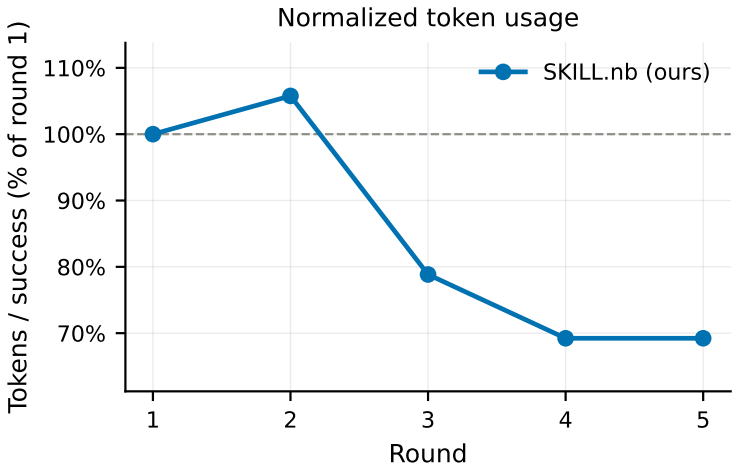


Figure 11: SKILL.nb recorded inference tokens per success across five rounds, including execution and maintenance, normalized to its round-1 value (100%).

B.4 GitLab Version Drift

We use controlled GitLab version drift because persistent web-agent artifacts can fail when a maintained application changes outside the agent's control. This better matches how web-agent artifacts can fail in practice than synthetic UI perturbations chosen by the experimenter, since the user's underlying work remains similar while selectors, workflows, and stored assumptions may become stale. The experiment therefore tests reuse across an application-version change without treating it as evidence for broader drift settings.

This appendix provides protocol support for the main GitLab version-drift experiment in Section 4.3. In Figure 2, v15, v16, and v18 abbreviate GitLab 15.7, GitLab 16.11, and GitLab 18.9, respectively. The candidate pool is the 180 single-site GitLab tasks from WebArena-Verified. We exclude multi-site tasks that include GitLab because those tasks add cross-site coordination and non-GitLab state, which would make the measured change less specific to GitLab application-version drift. The source deployment is GitLab 15.7. We evaluate GitLab 16.11 as an intermediate target and GitLab 18.9 as a longer-horizon target. We construct each target deployment by applying GitLab's official upgrade procedure[7] to the source deployment data, preserving the task data while changing the GitLab system version.

We selected these versions to test two levels of controlled drift. GitLab 15.7 matches the source environment for the benchmark tasks. GitLab 16.11 is the intermediate target because GitLab 16.0 introduced the new navigation experience and made the new Web IDE the default multi-file editor,[8] while GitLab 16.11 redesigned the project overview page so project information and links appear in a sidebar.[9] These changes overlap with the task mix: non-exclusive keyword grouping of the 180 single-site GitLab tasks gives about 40 issue-related tasks, 17 merge-request or review tasks, 20 file-editor or template tasks, and 132 project, repository, group, or member tasks. GitLab 18.9 is retained as a longer-horizon target after additional accumulated application changes.

B.4.1 Example UI Drift in Project Navigation

Figure 12 shows the same project overview across the three GitLab deployments. The red boxes mark the UI control used to open commit history. In GitLab 15.7, there is no dedicated history button on the project file toolbar. The agent must instead click the commit-count link in the project summary. In GitLab 16.11 and GitLab 18.9, GitLab exposes a dedicated `History` button, but its position changes as the project summary and action toolbar are reorganized. This example illustrates why

[7] https://docs.gitlab.com/update/upgrade/
[8] https://docs.gitlab.com/releases/16/gitlab-16-0-released/
[9] https://docs.gitlab.com/releases/16/gitlab-16-11-released/

old procedural state can become stale even when the underlying task intent and project data are unchanged.

The underlying DOM changes as well. GitLab 15.7 exposes the entry point as a project-stat link, roughly `a.nav-link.stat-link`, with a nested `svg[data-testid=commit-icon]` and visible text `48 Commits`. GitLab 16.11 exposes a button-style anchor with visible text `History`, but without a dedicated test identifier. GitLab 18.9 adds `data-testid=last-commit-history` to the corresponding anchor. This supports the qualitative observation in Section 4.3: newer GitLab versions can expose more stable DOM locators, and `data-testid` attributes are preferred by browser automation because they are less tied to layout or styling than class names.

Table 13: DOM-level differences for the same commit-history navigation target across GitLab versions.

Version	Observed DOM signature	Automation implication
GitLab 15.7	`a.nav-link.stat-link` with visible text `48 Commits`; no dedicated history button	Requires selecting a project-stat link rather than a toolbar action
GitLab 16.11	`a.btn` with visible text `History`; no `data-testid`	Adds the history action, but selectors still depend on class names or visible text
GitLab 18.9	`data-testid=last-commit-history` on the `History` anchor	Provides a stable test identifier preferred for automated browser locators

We define $\mathcal{D}_{\mathrm{mig}}$ as the subset of the 180 candidate tasks whose required task state and evaluator can be migrated or reconstructed on both target deployments. In the current migrated-run aggregate, $|\mathcal{D}_{\mathrm{mig}}|/180 = 180/180$. All success rates in this experiment are computed on $\mathcal{D}_{\mathrm{mig}}$.

We evaluate five conditions on the same $\mathcal{D}_{\mathrm{mig}}$ denominator. The source fresh condition starts each method from empty persistent state on GitLab 15.7. The two target fresh conditions start each method from empty persistent state on GitLab 16.11 and GitLab 18.9. The two old-to-target reuse conditions first start each method from empty persistent state on GitLab 15.7 and run on $\mathcal{D}_{\mathrm{mig}}$ to construct old-version state, then snapshot that state for target-version evaluation. For $\mathrm{AWM}_{\mathrm{online}}$, the snapshot is the induced workflow memory. For ReasoningBank, the snapshot is the distilled reasoning bank, memory pool, and retrieval index as used by the implementation. For SKILL.nb, the snapshot includes the workflow repository $\mathcal{K}$, workflow states, event logs needed by the lifecycle policy, and threshold and lifecycle state.

During target-version reuse evaluation, each target task restores the old-version snapshot or uses an isolated copy of that snapshot. Transient context, browser state, and environment state reset per task in all five conditions. Within-task fallback or local repair is allowed only when it is part of the method's normal execution. For SKILL.nb, gates, the code $\rightarrow$ NL procedure $\rightarrow$ intent cascade, local repair, and lifecycle correction remain active within each task. The restored snapshot fixes the starting persistent state while allowing these runtime responses to the migrated application. Any durable workflow, repository, memory, retrieval-index, event-log, threshold, or lifecycle update written during one GitLab 16.11 or GitLab 18.9 reuse task is discarded before later target tasks are scored. The fresh target-version conditions control for target-version difficulty, while the reuse conditions measure target-version success under old-state reuse rather than cumulative target-version relearning.

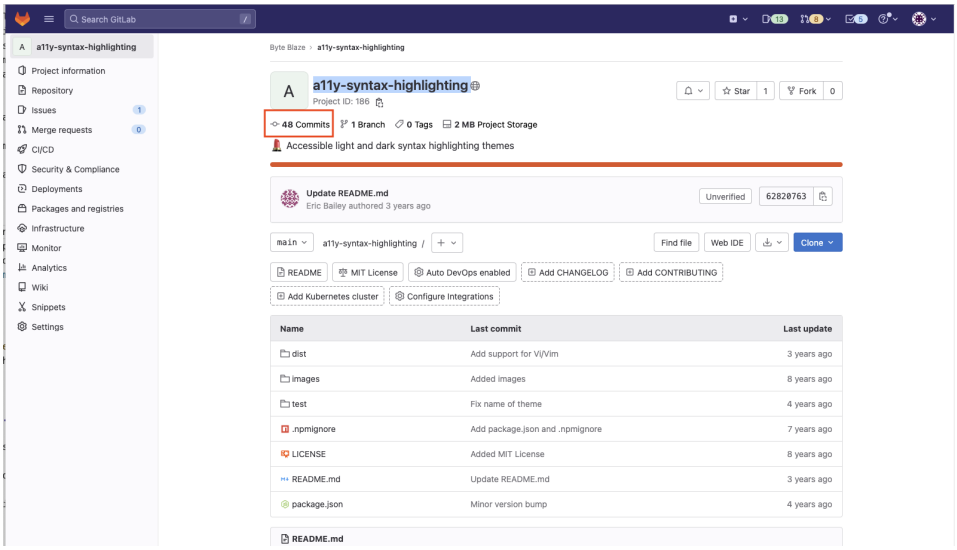
Table 14 gives the appendix results. The fresh columns measure performance when the method starts empty on each version. The reuse columns measure target-version performance when initialized from the GitLab 15.7 persistent-state snapshot.

For each success-rate cell and each point in Figure 2, we compute a two-sided 95% Wilson binomial confidence interval from the number of successful tasks k out of $n = |\mathcal{D}_{\mathrm{mig}}|$ migrated tasks. With $\hat{p} = k/n$ and $z = 1.96$, the interval is

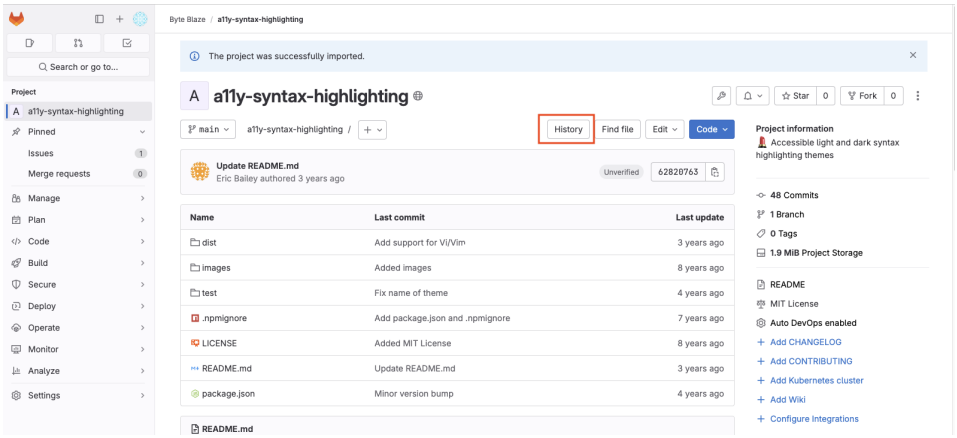
$$\frac{\hat{p} + z^2/(2n) \pm z\sqrt{\hat{p}(1-\hat{p})/n + z^2/(4n^2)}}{1 + z^2/n}.$$

We multiply the endpoints by 100 when reporting percentages.

(a) GitLab 15.7



(b) GitLab 16.11



(c) GitLab 18.9

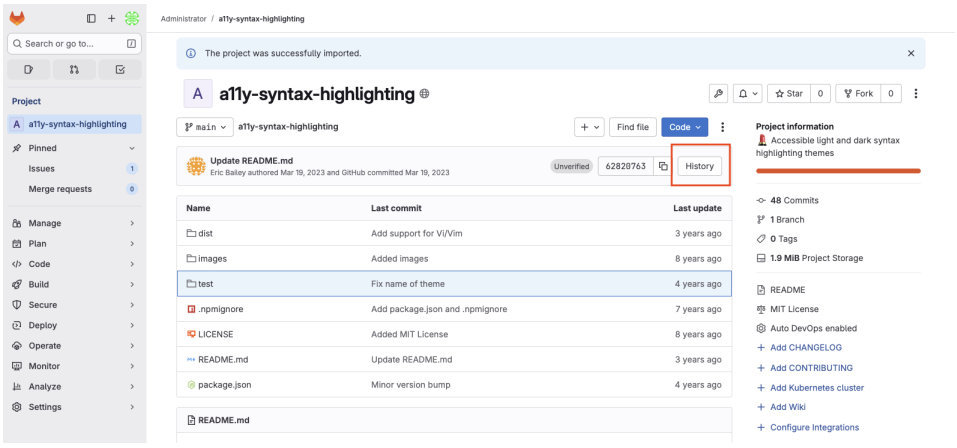


Figure 12: Example project-level UI drift across GitLab versions. The screenshots show the same project view with the commit-history entry point highlighted in red. GitLab 15.7 exposes commit history through the commit-count link, while GitLab 16.11 and GitLab 18.9 expose a dedicated `History` button in different toolbar positions.

Table 14: Appendix support table for the GitLab version-drift evaluation on $\mathcal{D}_{\mathrm{mig}}$. Each persistent-state method is evaluated under source fresh start, target fresh starts, and target-version reuse from a GitLab 15.7 persistent-state snapshot. Cells report SR with 95% Wilson confidence intervals.

Method	15.7 fresh	16.11 fresh	15.7→16.11 reuse	18.9 fresh	15.7→18.9 reuse
$\mathrm{AWM}_{\mathrm{online}}$	40.0 [33.1,47.3]	48.3 [41.1,55.6]	33.9 [27.4,41.1]	48.9 [41.7,56.1]	34.4 [27.9,41.6]
ReasoningBank	45.6 [38.4,52.8]	51.7 [44.4,58.9]	41.1 [34.2,48.4]	50.0 [42.8,57.2]	38.9 [32.1,46.2]
SKILL.nb	54.4 [47.2,61.6]	61.7 [54.4,68.5]	60.0 [52.7,66.9]	61.1 [53.8,67.9]	61.7 [54.4,68.5]

C Additional Experiments

C.1 Controlled Component Ablations

In this section, we include a component ablation to address whether the final SKILL.nb behavior depends on its main components rather than on a single incidental design choice. On the 258-task WebArena-Verified hard subset, the experiment examines selective code formalization with SKILL.nb$_{\text{NL-only}}$, which keeps only workflow intents and natural-language procedures; fallback with SKILL.nb$_{\text{code-only}}$, which disables fallback when executable code is unavailable or brittle; runtime validation with SKILL.nb$_{\text{no-gates}}$, which removes precondition and postcondition checks; and lifecycle cleanup with SKILL.nb$_{\text{no-cleanup}}$, which disables demotion and retirement together. All variants use the same model, harness, task stream, repository schema, evaluation budget, and online maintenance setting; each variant maintains its own repository state. Threshold specialization is evaluated separately in Appendix C.2.

Table 15 reports the resulting success, SKILL.nb-internal token cost, fallback, repair, and regression diagnostics.

Table 15: Controlled component ablations on the 258-task WebArena-Verified hard subset. All rows are SKILL.nb variants, so tokens per success use the same SKILL.nb-internal maintenance and update accounting and are reported in thousands. SR is reported as percent with 95% Wilson confidence intervals. Fallback is the percent of tasks using natural-language or raw-intent fallback; repair is the percent of tasks with an accepted repair or repository update; regression is update-induced failure after accepted repair or update.

Variant	SR (95% CI)	Tok. / succ.	Fallback	Repair	Regression
SKILL.nb	38.4 [32.6, 44.4]	36k	18.0%	73.0%	3.3%
SKILL.nb$_{\text{NL-only}}$	33.3 [27.9, 39.3]	57k	34.0%	61.0%	8.8%
SKILL.nb$_{\text{code-only}}$	31.0 [25.7, 36.9]	42k	–	48.0%	14.7%
SKILL.nb$_{\text{no-gates}}$	32.6 [27.1, 38.5]	45k	16.0%	66.0%	18.6%
SKILL.nb$_{\text{no-cleanup}}$	34.1 [28.6, 40.1]	47k	20.0%	67.0%	12.4%

SKILL.nb has the highest point SR and the lowest internal token cost per success. Removing code formalization increases cost and fallback use, removing fallback lowers SR and increases regression, and removing gates increases regression. Jointly disabling demotion and retirement lowers SR from 38.4% to 34.1% and raises regression from 3.3% to 12.4%; this comparison evaluates their combined contribution. Because this is a single-stream online ablation on the hard subset, we interpret the pattern as diagnostic evidence that the components are useful rather than as a benchmark-wide causal estimate.

C.2 Adaptive Threshold Ablation

In this section, we include a diagnostic ablation to isolate the contribution of the learned lifecycle policy from the workflow representation itself. Figure 13 reports this ablation on the 258-task WebArena-Verified hard subset over three rounds. It compares four SKILL.nb variants: the loose fixed variant uses one permissive fixed threshold vector shared across all groups and rounds, the strict fixed variant uses a more restrictive fixed vector, the pooled variant uses pooled thresholds from the offline replay estimator without group specialization, and SKILL.nb (ours) uses the full deployed group-specialized policy. In the current setup, the loose fixed vector is $(\tau_{\text{create}}, \tau_{\text{form}}, \tau_{\text{demote}}, \tau_{\text{retire}}) = (4, 3, 5, 0.50)$ and the strict fixed vector is $(8, 5, 2, 0.25)$. Panel (a) asks whether specialization improves task success, panel (b) whether ordinary lifecycle promotions introduce regressions on previously passing cached traces, and panel (c) whether those gains come with a lower maintenance-token measure relative to the loose fixed baseline at each round.

By round 3, SKILL.nb has the highest success rate (38.4%) and lowest update-induced regression (3.3%). Its reported round-3 maintenance-token measure is 67k versus 124k for loose fixed thresholds, or 54.0% of that baseline (panel (c)). The loose fixed variant's success falls from 32.0% to 27.1% while regression rises to 22.0%; the strict fixed variant remains near 31% success and has the lowest

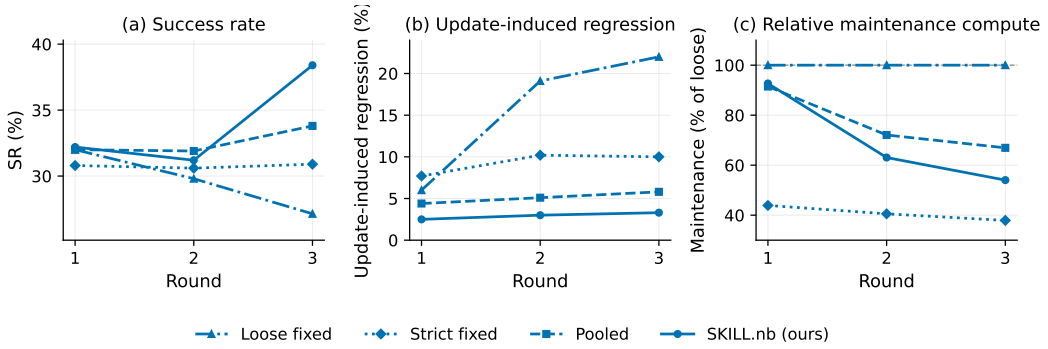


Figure 13: Diagnostic adaptive-threshold ablation on the WebArena-Verified hard subset (258 tasks) over three rounds. The four SKILL.nb variants differ only in how lifecycle thresholds are chosen: loose fixed thresholds, strict fixed thresholds, pooled thresholds from the offline estimator without group specialization, and the deployed group-specialized policy (SKILL.nb, ours). (a) Success rate by round. (b) Update-induced regression on previously passing cached traces, by round. (c) Maintenance-token measure at each round, normalized to the loose fixed variant at that round (loose fixed $= 100\%$).

maintenance measure in every round. At round 3, the pooled variant has higher success (33.8%) and lower regression (5.8%) than either fixed variant, but its maintenance measure (83k) exceeds strict fixed (47k). The group-specialized policy improves all three round-3 values relative to pooled thresholds. These comparisons describe the observed success–regression–maintenance trade-off; panel (c) reports the exported maintenance measure at each round, normalized to loose fixed. The brief round-2 dip for SKILL.nb mirrors the lifecycle result and is followed by the strongest round-3 recovery.

C.3 Gate-Incident Audit

We audited retained traces from rounds 1 and 5 of the five-round WebArena-Verified experiment, with 812 task executions per round. Task success was 436/812 in round 1 and 452/812 in round 5. The audit asks whether a gate accepted an invalid local state (a false positive) or rejected a valid local state (a false negative). The reference is the intended local step condition judged from the retained trace, not the hidden task oracle. The classification unit is the task's earliest detected attributable gate incident, so a cascade does not contribute multiple incidents to the table.

Extraction and review. The audit combines programmatic candidate extraction with focused human review. In round 1, 408 tasks with gate events contain approximately 4,900 checks. Candidate patterns include inconsistent pre/post pairs, gates that fail and then pass unchanged, and tasks that fail despite all gates passing. One reviewer examines candidate traces and labels the earliest attributable incident. All 812 tasks per round receive a category in Table 16, but the individual checks are not exhaustively hand-labeled. "No detected attributable gate error" includes failures attributed elsewhere and apparent clean successes; successes without repair or fallback were not manually screened for silent false positives.

Table 16: Task-level gate-incident audit. Percentages use all 812 task executions in each round, not the number of gate checks. The first three rows form a partition; the final row sums the two incident categories. These are detected task incidences, not gate-level false-positive or false-negative rates.

First detected incident	Round 1, count (%)	Round 5, count (%)
No detected attributable gate error	679 (83.6)	783 (96.4)
False positive: accepted invalid state	91 (11.2)	16 (2.0)
False negative: rejected valid state	42 (5.2)	13 (1.6)
Any detected gate incident	133 (16.4)	29 (3.6)

Failure modes and maintenance observations. In round 1, 58 of the 91 detected false positives involved incomplete predicates, while 32 of the 42 false negatives involved over-strict predicates. Post-gates accounted for 93 of the 133 first incidents and pre-gates for 40. Appendix A.5.1 gives two temporal examples. Case review records maintenance strengthening incomplete checks from element presence to the intended value, relaxing over-strict checks, broadening valid entry states, and dropping transient checks. The contribution of each type of edit was not separately quantified.

Execution and repository paths. Of the 376 round-1 task failures, 275 had no detected attributable gate error. Separately, the audit records 32 tasks recovered from a gate error through repair and 20 false negatives with no downstream effect; these diagnostic labels are not an additional disjoint partition, and their overlap was not reported. Among the 133 incident-bearing tasks, 85 produced no repository change (including 55 deferred to maintenance review), 45 led to demotion or retirement, and three reached promotion. These counts describe recorded paths, not the changes caused by gate errors: the audit does not establish which decisions would have differed without an incident, or whether the three promotions were harmful.

Interpretation and limits. Detected task incidence falls from 16.4% in round 1 to 3.6% in round 5. This comparison does not isolate the causal effect of maintenance: the artifacts and gate checks encountered can differ between rounds, and silent false positives can escape the review procedure. The audit therefore does not estimate gate-level precision or recall, establish semantic completeness, or measure the frequency of every gate error. It supplements the component ablation with observed failure modes and downstream paths under the stated detection procedure.

C.4 COLD-START EVALUATION: FIRST-REQUEST-ONLY SETTING

We evaluate task success and inference cost when each request must be handled without reusable state from earlier executions. The online benchmark (§4.1) allows workflow or memory state to accumulate across requests, while the reuse-consistency experiment (§4.2) tests artifacts retained from successful executions. In the first-request-only setting, we clear persistent state before every task so that each request starts without this history. This complements the online and reuse experiments by measuring the performance and inference cost of the within-task execution stack when cross-task reuse is unavailable.

Protocol. Before each request, SKILL.nb retains no workflow repository, event logs, or lifecycle-policy state, and ReasoningBank retains no reasoning bank. Browser and environment state reset per task as in the shared harness. SKILL.nb performs each task without pre-existing workflow gates, using observed environment changes to assess progress while recording a provisional workflow, Playwright traces, and OpenTelemetry events. Maintenance subsequently constructs missing gates from this evidence, as described in Appendix A.5.1. ReasoningBank retains its native ReAct scaffold and current-task history. Resulting durable state, including maintenance output, is discarded. Without accumulated calibration cases, automatic thresholded lifecycle actions defer to maintenance review; no artifact or policy history carries to the next request.

Table 17: Online and cold-start results on 812 WebArena-Verified tasks per method. SR is a percentage; brackets give marginal 95% Wilson intervals. Inference costs are thousands of recorded tokens per successful task, including maintenance inference.

	Success rate		Tokens/success	
Method	Online	Cold start (count; 95% CI)	Online	Cold start
ReasoningBank	49.8	46.4 (377; [43.0, 49.9])	44k	41k
SKILL.nb	53.7	48.8 (396; [45.3, 52.2])	52k	50k

Results and interpretation. Both methods execute all 812 tasks. SKILL.nb solves 396 versus 377 for ReasoningBank, a difference of 19 successes. Compared with the online runs, the counts decrease by 40 for SKILL.nb (436 to 396) and 27 for ReasoningBank (404 to 377). These results characterize complete native execution stacks under different persistence regimes; they do not isolate gates, threshold learning, or any individual lifecycle operation. We make no paired significance claim for the cold-start comparison; marginal interval overlap alone does not determine paired significance. CodeAct's stateless result in Table 1 is a contextual reference, not a separate cold-start run; $\text{AWM}_{\text{online}}$ is not included in this experiment.

Inference cost. Cold-start inference cost is 50k tokens per success for SKILL.nb and 41k for ReasoningBank, approximately a 22% premium. The accounting includes recorded execution and maintenance inference, including provisional construction and maintenance whose output is discarded; benchmark evaluation and post hoc analysis are excluded. The within-task paths remain method-specific, so this is a native-configuration cost comparison rather than a matched-compute test. It measures recorded inference cost per success for isolated requests under these configurations and does not establish lifecycle amortization for an isolated request.

C.5 Repository Storage Overhead

We provide an additional analysis of the storage overhead introduced by representing workflow repository entries as Jupyter notebooks and the text supplied to the agent when accessing a step. Notebook JSON includes cell structure, metadata, and outputs, which add overhead on disk. Its addressable cells also allow the agent to request a step's contents directly, without reading or searching the entire artifact. The notebook CLI parses serialization outside model context and returns the requested cells (Appendix A.4.2); this does not require the agent itself to parse notebook JSON. Thus, greater disk usage need not imply a larger model-context payload.

To quantify this tradeoff for one workflow, we convert a five-step GitLab notebook to Markdown with identical content and compare selected-cell notebook access with full-file Markdown activation. In the latter configuration, the agent receives the complete Markdown text even when it needs only one step. This analysis complements the recorded inference costs in Appendix B.3 by separating the artifact's disk footprint from the text supplied for discovery, successful step execution, and repair.

For a controlled comparison, we use an analytically constructed status representing normal successful completion of Step 1, with the workflow content held fixed across representations. This models the successful-step case encountered during execution while allowing us to compare the text returned by the two loading interfaces. The source artifact contains no screenshots, and text-token counts are estimated at four characters per token.

Table 18: Controlled payload comparison for one workflow using different access patterns over the same content. The analytically constructed status represents successful completion of Step 1. Text-token counts are estimates at four characters per token.

Payload	Full-file Markdown	Selected-cell notebook
Storage (bytes)	27,505	42,669
Discovery (estimated tokens)	382	382
Step-1 success (estimated tokens)	6,877	612
Step-1 repair (estimated tokens)	6,877	3,504

Table 18 shows the storage–access tradeoff. The notebook occupies 42,669 bytes versus 27,505 for Markdown, while both formats have the same estimated discovery-token count. For Step 1, direct cell access supplies an estimated 612 tokens in the success scenario and 3,504 in the repair scenario, compared with 6,877 for full-file Markdown activation in either case. The agent therefore receives the relevant step content with a smaller text payload despite the larger artifact on disk. This advantage comes from the selected-cell interface; a chunked or indexed Markdown client could provide similar selectivity.