Learning to Commit: Next-Commit Prediction via Online Supervised Contrastive Reflection

Mo Li[1,2] **Qitai Tan**[1] **Kai Chen**[2] **Ting Cao**[1†] **Yunxin Liu**[1,2]
[1]Tsinghua University [2]Shanghai AI Laboratory
limo.research@gmail.com

Abstract

Large language model (LLM)-based coding agents achieve strong results on controlled benchmarks yet routinely produce pull requests that real maintainers reject. The cause is not functional incorrectness but a lack of *organicity*: generated code ignores project conventions, duplicates internal APIs, and violates implicit architectural constraints. The latest repository snapshot is insufficient—it reveals the final codebase state, not the change patterns that shaped it. We make three contributions. **(1) Paradigm:** we propose **next-commit prediction** as a paradigm for learning agentic coding skill from a repository's own history—each historical commit is a self-supervised target whose oracle diff supplies dense supervision. **(2) Evaluation:** we establish **Organicity Evaluation** as a measurable objective for coding agents and contribute the **Learning to Commit benchmark**—5 curated open-source GitHub repositories under strict per-repository temporal splits—scoring patches along file localisation, internal API reuse, patch bloat, and code-style consistency. **(3) Method:** we instantiate the paradigm with the **Learning to Commit** framework, in which the agent performs **supervised contrastive reflection**: it blindly attempts each historical commit, contrasts its prediction against the oracle, and incrementally distils a reusable *skill* document that conditions subsequent generations. On the Learning to Commit benchmark and on SWE-bench Pro reframed under our temporal-split protocol, our framework consistently improves organicity on held-out future tasks and further lifts test-pass rates, narrowing the gap between benchmark success and mergeability.

1 Introduction

The past three years have witnessed a remarkable acceleration in AI-assisted software engineering. Coding agents built on large language models now resolve a substantial fraction of tasks on curated benchmarks such as SWE-bench [1], while libraries like HumanEval [2] and MBPP [3] are largely saturated. Repository-scale benchmarks such as FEA-Bench and FeatureBench further extend evaluation from isolated code generation to multi-file feature implementation in realistic repositories [4, 5]. These numbers have fuelled optimism that LLM agents are ready for industrial deployment. Yet professional maintainers of complex repositories remain cautious: they acknowledge that AI-generated code is often functionally correct, but still routinely reject pull requests because the code feels **alien**—written by someone who has never lived through its history. In practice, this alien quality manifests not only as stylistic mismatch but also as **unnecessary code bloat**: the agent re-implements utilities, wrappers, or control flow patterns that the repository already contains.

The gap between benchmark scores and industrial merge-rate reflects a structural blind spot in how we currently evaluate coding agents. Benchmarks such as SWE-bench still cast software engineering as a sequence of isolated, one-off tasks: an agent sees an issue, edits the codebase, and

[†]Corresponding author.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

Table 1: Comparison of representative repository-level benchmarks and agentic learning paradigms. Columns progress from basic capabilities to our unique contributions: leveraging historical context, continual learning across tasks, strict data leakage prevention, oracle-supervised memory updates, and multi-dimensional organicity evaluation. ✓ = satisfied; ✗ = not satisfied.

Framework / Paradigm	Chronological Eval	Historical Memory	Leakage Prevention	Oracle Supervision	Organicity Eval
SWE-bench [1]	✗	✗	✗	✗	✗
SWE-Bench++ [18]	✗	✗	✓	✗	✗
RepoMem [8]	✗	✓	✗	✗	✗
SWE-CI [7]	✓	✗	✗	✗	✗
SWE-Bench-CL [13]	✓	✓	✗	✗	✗
Learning to Commit (Ours)	✓	✓	✓	✓	✓

succeeds if the tests pass, without carrying any accumulated knowledge forward. Furthermore, they typically evaluate issues out of chronological order, ignoring how a repository and its conventions naturally evolve over time. COMMIT0 [6] stresses whole-library implementation from API specifications, while SWE-CI [7] extends evaluation over months of consecutive commits. At the same time, repository-level systems such as Repository Memory and SGAgent show that richer codebase context improves localisation and repair [8, 9], and PR-centric works such as Coeditor, Clean-PR, and R2E-Gym suggest that commit and PR histories are valuable learning signals [10–12]. Yet these settings still share the same basic premise: the agent is not expected to undergo anything like repository-specific onboarding before it starts writing code. This omission matters because the current repository snapshot shows only the finished building, not why particular supports, interfaces, and boundaries were introduced along the way. More broadly, it reflects a shift that is now becoming central in agent research: capability no longer depends only on managing context within a single session, but increasingly on managing and updating memory across sessions and across time. Two prevailing paradigms attempt to address this need but face structural limitations: sequential reformulations of static benchmarks [13–15]—across heterogeneous tasks, what should plausibly transfer between examples is unclear, and unsupervised self-reflection without ground-truth oracles accumulates self-reinforcing errors; and personalised memory benchmarks [16, 17] targeting narrow conversational recall, **with LongMemEval increasingly saturated by frontier long-context models** (see Appendix C).

The gap between benchmark success and real-world merge rates fundamentally stems from a lack of repository memory. When core maintainers resolve an issue, they draw upon their lived experience of the project's evolution, intuitively leveraging past commits, module boundaries, and idiomatic utilities. This accumulated memory ensures their pull requests grow organically from the repository. In contrast, a newly joined junior engineer—or a stateless AI agent—lacks this historical context. Current agents operate exactly like these amnesiac newcomers, producing what we term **alien code**: functionally correct but stylistically foreign, architecturally dissonant, and bloated with redundant reimplementations. Maintainers reject such patches for the exact same reasons they would ask a newcomer to revise a historically disconnected first contribution.

To generate code that is not merely correct but genuinely **organic**, we introduce **Learning to Commit**, a framework centred on Online Repository Memory. The key idea is simple: organicity is learnable from the commit history, which records how a project has chosen to evolve over time. We frame this as a problem of **next-commit prediction**: each historical commit serves as a self-supervised target whose prediction is conditioned on the repository's prior evolutionary state. Given a repository with a strict chronological split, the agent performs **supervised contrastive reflection** on earlier commits—blindly attempting each historical change, comparing its prediction against the oracle diff, and distilling the gap into a continuously growing, incrementally updatable set of **skills**. When a new PR description arrives, the agent retrieves and conditions its generation on these skills, producing changes aligned with naming conventions, abstractions, and maintainer preferences.

Evaluating coding agents is frequently compromised by pre-training data leakage—a pervasive vulnerability in current evaluations, as evidenced by recent efforts like SWE-Bench++ [18]. By design, our same-repository, time-split evaluation prevents train-test memory leakage during onboarding: skills are built exclusively from commits before a hard cutoff date, while evaluation tasks are drawn from genuinely future merged commits. In this paper, we establish repository-personalised adaptation as a first-class evaluation objective for coding agents. Our contributions are:

- We propose **next-commit prediction** as a real-world online-learning paradigm for coding agents: grounded in repository evolution, densely supervised by oracle diffs, and evaluable along multiple **organicity dimensions**.
- We instantiate this paradigm with the **Learning to Commit** framework, centred on **supervised contrastive reflection**: the agent blindly attempts each historical commit, contrasts its prediction against the oracle diff, and distils skills that condition later generations.
- We construct and contribute a strict time-split benchmark of curated repositories with multi-dimensional metrics, demonstrating that our framework effectively improves organicity over existing paradigms; further validation on SWE-bench Pro [19] shows additional pass-rate gains on a public, contamination-resistant testbed.

2 Related Work

2.1 Static Evaluation Paradigms and Repository-Level Agents

Early benchmarks evaluated large language models in isolated, stateless environments [2, 3], which recently evolved into realistic, repository-level tasks like SWE-bench [1] and its multi-file or generative extensions [4–6]. To tackle these complex scenarios, recent works scale up training data through automated PR harvesting [11, 12, 20], with works like SWE-Bench++ [18] specifically designing recent cutoffs to ensure zero data leakage and design multi-agent workflows with specialized roles and reasoning steps [9, 21]. However, these data-centric and workflow-based approaches treat tasks as static snapshots, completely ignoring the chronological evolution of software. While SWE-CI [7] introduces a temporal CI-loop to penalize technical debt, it primarily evaluates performance degradation rather than empowering the agent to actively learn and internalize repository-specific conventions over time. Crucially, these existing paradigms rely almost exclusively on functional test passes as their success metric, lacking multi-dimensional organicity evaluation to assess codebase stylistic consistency, internal API reuse, and architectural fit.

2.2 Agent Memory and Continual Learning in Software Engineering

To capture historical developer intent, models have incorporated commit histories via static weight updates [10, 22] or passive retrieval augmentation [8], yet both lack an active trial-and-error learning process. The necessity for dynamic continual learning is highlighted by the performance degradation observed in long-horizon tasks [7, 23], prompting explorations into online agent memory at various granularities [24–26]. Most notably, a series of works evaluate continual learning by reformulating static datasets into sequential task streams [13–15]; yet across heterogeneous tasks it is unclear what memory should plausibly transfer between examples, and unsupervised self-reflection without ground-truth oracles is highly vulnerable to self-reinforcing errors [24]. A separate line of personalised memory benchmarks [16, 17] primarily targets narrow conversational and episodic recall—a setting increasingly saturated by frontier long-context models (Appendix C). Even with stepwise environmental feedback [27], signals remain too noisy to extract deep design patterns. Our work bridges this gap: by strictly partitioning history to prevent data leakage, our agent attempts historical commits and receives *oracle diffs* as dense supervision. Through supervised contrastive distillation between blind attempts and expert patches, the agent extracts reusable, repository-specific development patterns into a continuously refined skill document, ensuring failures drive optimal, history-conditioned memory updates. Table 1 summarises how our method compares with representative prior benchmarks across these key dimensions.

3 Methodology

As illustrated in Fig. 1, the Learning to Commit framework instantiates next-commit prediction by decomposing the agent's interaction with each repository into two phases: (1) **Repository Onboarding** (§3.2), where the agent iteratively builds a reusable skill document $\mathcal{M}$ from the history prefix $\mathcal{C}^-$ through on-policy contrastive reflection; and (2) **Skill-Conditioned Resolution** (§3.3), where, given a future task description d_k and the corresponding repository snapshot $\mathcal{S}_k$, the agent autonomously generates a patch conditioned on the accumulated skills $\mathcal{M}$.

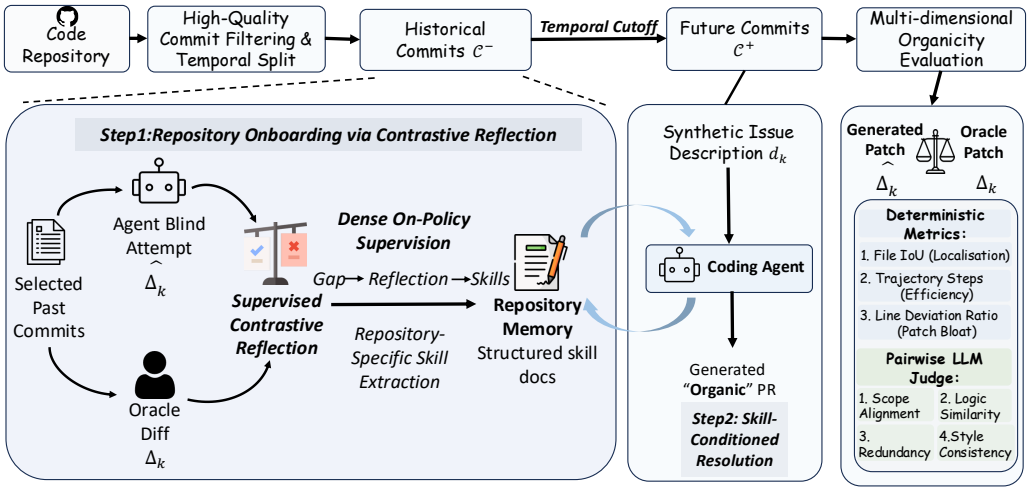

Figure 1: Overview of the Learning to Commit framework. Repositories are time-split into past commits (skill learning) and future commits (evaluation). The agent extracts reusable skills via on-policy contrastive reflection, conditions on the accumulated skills to generate organic patches for future tasks, and is scored against oracle commits along multiple organicity dimensions.

3.1 Problem Formulation

Let $\mathcal{C} = (c_1, c_2, \ldots, c_T)$ denote a chronological sequence of high-quality commits from a target repository $\mathcal{R}$. A temporal cutoff T^* partitions the sequence into a history prefix $\mathcal{C}^- = \{c_t : t \leq T^*\}$ for learning and a test set $\mathcal{C}^+ = \{c_k : k > T^*\}$ for evaluation. For each test commit $c_k \in \mathcal{C}^+$, the agent observes the repository snapshot $\mathcal{S}_k$ and a synthetic issue description d_k, and must produce a patch $\hat{\Delta}_k$ approximating the oracle next-commit diff:

$$\hat{\Delta}_k \;=\; \mathcal{A}\big(d_k,\, \mathcal{S}_k \,;\, \mathcal{C}^-\big)\,, \tag{1}$$

where $\mathcal{A}$ denotes the agent and $\mathcal{C}^-$ is the only available adaptation signal. This sequential formulation provides a real-world learning paradigm grounded in genuine project history.

3.2 Repository Onboarding via Contrastive Reflection

The learning phase mimics how a human developer onboards onto a new project: not by passively reading documentation, but by actively attempting tasks and learning from the gap between one's own output and expert practice. We initialise an empty skill document $\mathcal{M}^{(0)} = \varnothing$. For each learning commit $c_t \in \mathcal{C}^-$, the agent executes a three-step loop. **Step 1 (Blind Attempt):** The agent receives the repository snapshot $\mathcal{S}_t$ and the synthetic issue description d_t, along with the current skill document $\mathcal{M}^{(t-1)}$. It autonomously explores the codebase using standard tool-use capabilities (file reading, searching, editing; full tool inventory and per-phase usage in Appendix I) and produces a candidate patch $\hat{\Delta}_t$. **Step 2 (Oracle Revelation and Contrastive Reflection):** The ground-truth oracle diff Δ_t—representing the accepted solution by a human domain expert—is revealed. The agent compares its own attempt $\hat{\Delta}_t$ against Δ_t, identifying discrepancies in file localisation, implementation logic, API usage, and coding style. The gap between the two serves as dense, on-policy supervision: the larger the discrepancy, the richer the learning signal. Because $\hat{\Delta}_t$ is the agent's own current-policy rollout rather than a record from a static dataset, this supervision is *on-policy* in the RL sense, targeting the agent's intrinsic capability gaps rather than generic errors. **Step 3 (Skill Update):** Based on the contrastive reflection, the agent updates the skill document through explicit CRUD operations—creating new entries for previously unknown patterns, revising entries that were partially correct, and deprecating entries contradicted by the oracle evidence:

$$\mathcal{M}^{(t)} = \textsc{Update}\Big(\mathcal{M}^{(t-1)},\, \hat{\Delta}_t,\, \Delta_t,\, d_t\Big)\,. \tag{2}$$

The resulting skill document $\mathcal{M}$ consolidates abstract, reusable development patterns that typically encompass: (1) coding style and naming conventions, (2) the existence and correct usage of internal API utilities, (3) implicit architectural constraints and module boundaries, and (4) maintainer preference patterns such as error handling style and test organisation. Unlike static RAG over commit histories, this on-policy learning loop ensures the extracted patterns are precisely calibrated to the agent's own capability gaps—addressing exactly the mistakes the agent would otherwise make.

3.3 Skill-Conditioned Resolution

When a future task $d_k \in \mathcal{C}^+$ arrives, the agent receives the repository snapshot $\mathcal{S}_k$, the task description d_k, and the full accumulated skill document $\mathcal{M}$. The agent then autonomously resolves the task using standard tool-use capabilities—reading relevant files, searching the codebase, and applying edits—conditioned on the development patterns recorded in $\mathcal{M}$. No rigid retrieval pipeline or preplanned workflow is imposed; the agent decides which skills to consult and which files to explore based on its own judgement, mirroring how a human developer with internalised project knowledge would approach a new task.

3.4 Dataset Construction

We construct the **Learning to Commit benchmark** by applying a unified data curation pipeline to a curated set of high-quality open-source GitHub repositories. The pipeline proceeds in five stages and is applied independently to each repository, preserving per-repository temporal splits. (i) **Commit scanning and quality filtering**: we extract each repository's full non-merge commit history, apply programmatic pre-filters to remove trivial changes (e.g., fewer than 10 modified lines, version bumps), and use an LLM to assess whether each remaining commit exhibits substantive, learnable development patterns. Commits whose diffs exceed 180K tokens are excluded (per-repository numbers in §4). (ii) **Unsupervised category clustering**: to prevent the learning curriculum from collapsing onto a single change pattern, we sample the rationales from the first stage and perform unsupervised clustering to identify core development categories spanning architecture design, concurrency, testing, etc. (iii) **Category tagging**: each retained commit is then assigned a primary category label by an LLM that considers the commit title, the rationale, and the full patch content.

Following the initial tagging, we construct the final benchmark sets. (iv) **Stratified sampling with temporal split**: within each repository, commits are chronologically ordered and partitioned into a learning prefix and a held-out test suffix. We perform stratified proportional downsampling across categories within each pool to construct a balanced per-repository curriculum. (v) **Synthetic query generation**: for each task, an LLM synthesises an issue-style natural-language query from the commit message and diff. The prompt specifies the "what" and "why" of the task while strictly omitting implementation details (exact file paths, function names, or solution strategies), emulating a realistic user issue that does not leak the oracle patch. Crucially, the strict per-repository temporal split ensures that all learning commits predate all test commits within each repository, completely preventing information leakage from evaluation tasks into the skill-building phase. Across all LLM-based stages above, we adopt a two-stage decoding scheme—free-form reasoning followed by a separate structured-extraction call, as direct structured generation consistently degraded quality in our pilot studies, a phenomenon also documented as the *format tax* [28].

3.5 Evaluation Metrics

Evaluating patch organicity requires moving beyond functional test passes. We formally define organicity as how indistinguishable a generated patch is from a patch written by a long-term maintainer of the repository. We operationalise this definition across four practical dimensions: (D1) localisation fidelity, editing the exact locations a maintainer would touch; (D2) solution-pattern consistency, utilising the repository's official extension points and idiomatic abstractions; (D3) change economy, avoiding redundant reimplementations and scope creep; and (D4) surface-convention adherence, matching the repository's naming, style, and test organisation. We evaluate generated patches $\hat{\Delta}_k$ against oracle patches Δ_k through two complementary metric families mapped to these dimensions.

Deterministic code metrics. We define two quantitative metrics serving as auxiliary diagnostics for D1 and D3 respectively, rather than as proxies for correctness. File IoU measures the Jaccard

similarity $|\mathcal{F}(\hat{\Delta}) \cap \mathcal{F}_k^*|/|\mathcal{F}(\hat{\Delta}) \cup \mathcal{F}_k^*|$ between the files modified by the agent and those in the oracle, quantifying localisation fidelity (D1). Line deviation ratio computes $\big|\,|\hat{\Delta}_k| - |\Delta_k|\,\big| \,/\, |\Delta_k|$, where $|\cdot|$ denotes the diff line count, targeting change economy (D3) as agents failing to reuse internal APIs typically produce inflated diffs. We additionally record trajectory steps, the number of tool-using assistant turns before the agent's `submit` action, to describe exploration depth.

Multi-dimensional LLM judge. For broader coverage across the dimensions, we employ a pairwise A/B evaluation protocol where an LLM judge compares the baseline agent against the skill-conditioned agent. The judge evaluates logic similarity corresponding to D2, redundancy mapping to D3, code style capturing D4, and scope alignment. Each task is judged across a four-round ABBA schedule to control for position bias, with verdicts averaged across rounds and dimensions. The correlation of this proxy suite with genuine human merge judgements is validated via a dual-track human study in Appendix D (on Track B, File IoU achieves a patch-level Spearman correlation of 0.40 with human merge verdicts, line deviation achieves -0.28, and the pairwise judge agrees with human preference on 21 out of 30 tasks).

4 Experiments

In this section, we evaluate the **Learning to Commit** framework to answer a series of research questions. We first introduce our evaluation benchmarks and agent configurations (§4.1). We then quantify the organicity of the generated commits (patches), comparing our supervised contrastive reflection against passive and unsupervised memory baselines, scanning across agent models, and ablating the learning mode (§4.2). Finally, we verify whether these organicity gains translate to functional correctness on SWE-bench Pro, and probe whether skill conditioning raises the ceiling (§4.3).

4.1 Experimental Setup

Datasets and benchmarks. We evaluate our framework across two distinct settings. To conduct multi-dimensional organicity evaluation, we construct the **Learning to Commit benchmark**, a curated suite of 5 high-quality open-source repositories from the AI engineering ecosystem, collectively spanning $63{,}417$ non-merge commits over 2.6–7.1 years of project history (selection criteria, sub-domains, and per-repository metadata in Appendix E). We apply the data curation pipeline described in §3.4 to each repository independently. Per repository, we perform stratified per-category sampling within the chronologically split learning and test pools to obtain 10 historical learning commits and 10 held-out test commits, yielding 100 commits across 5 repositories. The strict per-repository temporal split guarantees that all learning commits predate all test commits within each repository, preventing leakage from evaluation tasks into the skill-building phase. To evaluate functional correctness on a public, contamination-resistant testbed, we further utilise SWE-bench Pro [19]: for each test task at commit t, only commits strictly before t are eligible as onboarding material, preventing post-t commits—which may already encode the test fix—from leaking into the skill-building phase; full details of how we adapt the public SWE-bench Pro release to our temporal-split protocol are documented in Appendix F. A four-part contamination audit covering both settings is provided in Appendix G.

Agent configuration and baselines. We hold the agent loop and tool-use schema fixed across every condition; the only quantities we vary are (i) the agent **model**, (ii) the **memory-extraction method** that feeds the agent at solve time, and (iii) the **learning mode** used by our framework when extracting skills. The skill-conditioned agent receives the accumulated memory $\mathcal{M}$ produced by our framework; the baseline (vanilla) agent operates without any external memory.

We compare the following memory-extraction methods, all evaluated on the same Learning to Commit benchmark: **Vanilla agent** — no memory, the agent reads only the issue and the snapshot. **In-context history (passive RAG over commits)** — at solve time we embed the synthetic issue, retrieve the top-3 most similar historical commits (commit message + diff) from the same repository, and concatenate them into the prompt. **Unsupervised self-reflection** — the same online learning loop as ours, but with the oracle diff withheld: the agent attempts each historical commit, then summarises "what worked" from its own attempt and writes that into the skill document. **Observation-only reflection** — receives the task, the oracle diff, and the repository snapshot, and directly distils

Table 2: Three ablations on the Learning to Commit benchmark. The Claude Opus 4.6 + ours + `seq-all` configuration is the shared reference and appears once in each block. Pairwise win rate is computed against the matched no-skill / Vanilla reference (see Appendix I.4 for the exact formula); "—" marks reference rows. Bold marks rows that strictly improve over the matched reference on a given metric; a bold Pairwise indicates a strict $> 50\%$ win rate.

Configuration	**File IoU** (↑)	**Steps**	**Line Dev** (↓)	**Pairwise** (↑)
Different agent models (method: ours; learning mode: `seq-all`)				
Kimi K2.5 (w/o skill)	53%	30	0.65	—
Kimi K2.5 (w/ skill)	**60%**	45	**0.50**	**55%**
Claude Sonnet 4.6 (w/o skill)	58%	28	0.61	—
Claude Sonnet 4.6 (w/ skill)	**65%**	42	**0.46**	**57%**
Claude Opus 4.6 (w/o skill)	62%	26	0.58	—
Claude Opus 4.6 (w/ skill)	**67%**	40	**0.43**	**59%**
Different memory-extraction methods (model: Claude Opus 4.6; learning mode: `seq-all`)				
Vanilla agent (no memory)	62%	26	0.58	—
In-context history (passive RAG, top-3)	**64%**	28	**0.50**	**53%**
Unsupervised self-reflection (no oracle)	60%	35	0.62	44%
Observation-only reflection (no blind attempt)	**63%**	38	**0.46**	**58%**
Ours (supervised contrastive reflection)	**67%**	40	**0.43**	**59%**
Different learning modes (model: Claude Opus 4.6; method: ours)				
`seq-all`	**67%**	40	**0.43**	**59%**
`par-bycat`	**66%**	43	**0.45**	**57%**
`par-all`	**67%**	45	**0.48**	**53%**
`seq-bycat`	60%	47	0.55	42%

skills from the expert patch without a blind attempt. **Supervised contrastive reflection (ours)** — the loop in §3.2, which contrasts the blind attempt against the oracle diff before updating $\mathcal{M}$.

For agent models we report Kimi K2.5, Claude Sonnet 4.6, and Claude Opus 4.6 [29]. For learning modes we cross two curriculum strategies—*Sequential* (chronological in-place refinement of a single skill document) vs. *Parallel* (independent per-commit skills merged afterward)—with two assignment schemes—*all* (one shared skill over the whole corpus) vs. *by_category* (a separate skill per category)—yielding four configurations: `seq-all`, `seq-bycat`, `par-all`, and `par-bycat`. Running every (model, method, mode) cell jointly is cost-prohibitive. We therefore report each axis as a separate ablation against a fixed reference point (Claude Opus 4.6 + ours + `seq-all`). For SWE-bench Pro evaluation, we apply this same reference configuration. All conditions share a hard limit of 300 tool steps per phase as an implementation constant, though no condition approached this limit in practice.

4.2 How Organic Are the Generated Commits?

Table 2 reports the three ablations on the Learning to Commit benchmark, evaluated using the metrics defined in §3.5: file-level localisation (File IoU ↑), trajectory length, patch bloat (line deviation ↓), and pairwise win rate (↑, against the matched no-skill / Vanilla reference; "—" marks reference rows). The shared reference (Claude Opus 4.6 + ours + `seq-all`) sits at the intersection of all three ablations and appears once in each block.

Does supervised contrastive reflection beat alternative memory-extraction methods? The middle block of Table 2 compares our method against natural memory-extraction baselines. **Passive RAG over commits** retrieves the three most similar historical commits and prepends them to the prompt at solve time. It produces a small improvement over Vanilla, taking File IoU from 62% to 64%, because the agent occasionally lifts a relevant snippet wholesale. Yet RAG cannot abstract: when the test issue requires the same internal utility used differently, retrieved diffs become misleading boilerplate. **Unsupervised self-reflection** acts as the same online learning loop as ours but with the oracle diff withheld. Without ground truth to anchor reflection, the skill document accumulates self-reinforcing errors such as hallucinated APIs and redundant code, and the pairwise win rate drops to 44%, meaning unsupervised reflection is actively harmful. We further introduce an **observation-**

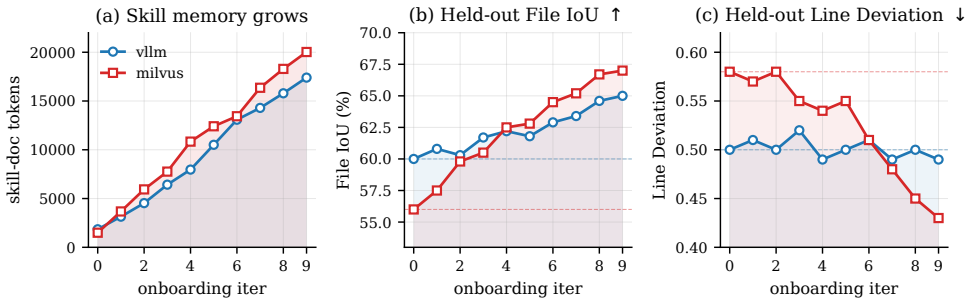

Figure 2: Trends across the online learning stage, evaluated on the held-out test commits of two benchmark repositories (vllm and milvus) from the Learning to Commit benchmark **(a)** Skill-document size in tokens, measured at every onboarding iter. **(b)** Held-out File IoU averaged over the 10 test commits per repo. **(c)** Held-out Line Deviation, same averaging.

only reflection baseline that skips the blind attempt: it receives the task, the oracle diff, and the repository snapshot, and directly distils skills from the expert patch. It extracts clean conventions, reaching a line deviation of 0.46 and a pairwise win rate of 58%, approaching our performance. However, tracing the localisation improvements reveals an ablation ladder: passive RAG improves File IoU by 2.0 percentage points, observation-only distillation improves it by 1.0 percentage point, while distillation coupled with on-policy blind attempts improves it by 5.0 percentage points. This indicates that on-policy contrast unlocks the localisation gain: blind attempts allow the agent to learn exactly where it naturally makes mistakes. Our supervised contrastive reflection outperforms all baselines, demonstrating that the oracle diff paired with on-policy execution provides a signal beyond what retrieval, unsupervised reflection, or observational learning can deliver.

How do gains scale across different models? The first block of Table 2 scans three agent models (Kimi K2.5, Claude Sonnet 4.6, Claude Opus 4.6) under the same method (ours) and learning mode (`seq-all`). Two patterns are clear. (i) The skill lift is consistent: every model improves on file localisation, patch tightness, and pairwise win rate against the matched no-skill baseline (+5 to +7 pp on File IoU, with Line Deviation reduced by roughly a quarter), while skill-conditioned agents take modestly more steps—a signal that they explore the repository more thoroughly before committing to a patch rather than a regression in efficiency. (ii) Stronger models not only end up with stronger absolute organicity but also produce tighter, less bloated patches, suggesting that the bottleneck without skills is repository-specific knowledge rather than reasoning capability—once a stronger model is given the right reusable patterns, it can deploy them more decisively. The framework is not parameterised solely by the headline model: gains carry through from Claude Opus 4.6 to lighter models like Claude Sonnet 4.6 and Kimi K2.5 as well.

How does the learning mode affect skill quality? The last block of Table 2 fixes the Claude Opus 4.6 model and our method, and varies the four learning configurations. Sequential learning with full-corpus assignment (`seq-all`) produces the highest-quality skills: File IoU reaches 67%, Line Deviation is the lowest at 0.43, and pairwise win rate is the highest at 59%. This is consistent with our expectation that progressive, on-policy refinement of a single skill document yields more coherent and deeply fused knowledge than parallel extraction followed by post-hoc merging. `par-bycat` comes a close second (File IoU 66%, Line Deviation 0.45, pairwise 57%), suggesting that focused, domain-aligned learning can largely compensate for the lack of sequential fusion. The worst configuration (`seq-bycat`) loses both ways: per-category sequential refinement starves each category of cross-category context, and the 42% pairwise win rate indicates that the resulting skill is actively misleading on a non-trivial fraction of tasks.

Where do the gains from skills actually come from? Our dimensional-level analysis of the Opus 4.6 runs, detailed in Table 10 of Appendix E, reveals that the gains from skill conditioning are concentrated on a few capability dimensions rather than spread evenly. The strongest gains appear in *core logic reproduction* (Q2 win rate 49.5% vs. 23.5% on the seq-all reference) and *redundancy reduction* (Q3 favours skill agents in three out of four configurations). Meanwhile, scope alignment (Q1) and code style (Q4) show only marginal or neutral effects. This pattern suggests that

Table 3: Overall pass@1 and organicity metrics on SWE-bench Pro across three agent models. pass@1 cells report mean ± 1 standard deviation (SD) across 4 independent agent rollouts; Δ is the per-run paired difference averaged across rollouts. File IoU and Line Deviation are computed against the oracle commit from the same trajectories.

Configuration	pass@1	Δ (pp)	File IoU	Line Dev
Kimi K2.5				
Baseline	$31.0_{\pm 2.6}\%$	—	53.2%	0.54
+ Skill (ours)	**34.0**$_{\pm 2.8}$**%**	**+3.0**$_{\pm 2.0}$	**58.4%**	**0.49**
Claude Sonnet 4.6				
Baseline	$34.0_{\pm 2.3}\%$	—	55.4%	0.51
+ Skill (ours)	**40.0**$_{\pm 3.3}$**%**	**+6.0**$_{\pm 2.3}$	**61.5%**	**0.47**
Claude Opus 4.6				
Baseline	$48.0_{\pm 2.8}\%$	—	58.1%	0.47
+ Skill (ours)	**56.0**$_{\pm 1.6}$**%**	**+8.0**$_{\pm 3.3}$	**65.5%**	**0.44**

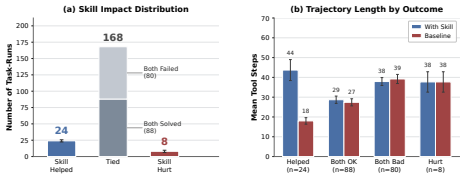

Figure 3: Per-task skill impact on the SWE-bench Pro subset (Claude Opus 4.6), pooled across 4 independent agent rollouts (200 task-runs total). **(a)** Outcome distribution by category (helped / tied / hurt); error bars on the helped/hurt bars show the run-to-run standard deviation of the pooled total. **(b)** Mean tool-call trajectory length per outcome group; error bars show ± 1 standard error of the mean across pooled per-task trajectories within each outcome group.

accumulated skills primarily help agents reproduce the correct core logic and avoid redundant or hallucinated code, while fine-grained scope alignment and code style are already largely inferable from the current repository snapshot and thus leave little headroom for commit-level skill accumulation to add value.

How do organicity metrics evolve across the online learning stage? Figure 2 reports three measurements on vllm and milvus across the 10 onboarding iterations: (a) skill-document size grows roughly tenfold; (b) and (c) held-out File IoU and Line Deviation averaged over test commits per repo. Both repositories show monotone File IoU gains, with milvus—which starts further behind—improving more sharply than vllm. On Line Deviation, however, the two repositories exhibit different trends: milvus drives a steady reduction while vllm remains largely stable. This suggests that skill conditioning primarily addresses the specific organicity dimensions where a codebase is initially weak, and yields marginal improvements when the baseline performance is already strong.

4.3 From Organicity to Functional Correctness: SWE-bench Pro

To test whether the organicity gains observed on the Learning to Commit benchmark translate into functional correctness, we evaluate on SWE-bench Pro [19]—a contamination-resistant public testbed that, unlike our self-curated benchmark, ships with human-quality executable test suites. We perform stratified sampling based on repository quotas to select 50 tasks; for each task at commit t, we apply programmatic pre-filters to the same repository's pre-t history and use an LLM to select the top-3 most relevant historical commits, presenting them to the agent in chronological order during onboarding. The agent model, tool-use capabilities, and prompts are held fixed across the with-skill and baseline conditions; only the skill document differs. Table 3 reports the overall pass@1 with versus without skill conditioning. Appendix F details the full subsampling protocol and the learning-commit-selection pipeline.

How do skills lift the ceiling, at what cost, and where do they hurt? Figure 3 dissects the Claude Opus 4.6 results at the per-task level, pooled across 4 independent rollouts. Because SWE-bench Pro evaluates general software engineering rather than explicitly targeting continuous learning, a high baseline of ties is structurally expected. Against this backdrop, skill conditioning yields a stable and replicable net positive lift across rollouts (Table 3). Examining the organicity metrics in Table 3, File IoU increases by roughly 5 to 7 percentage points across all three models while Line Deviation consistently decreases. This mirrors the magnitude observed on our main benchmark. The mechanism for the localisation gain is that while the task text anchors the core changes so both conditions find the main files, the oracle commit often edits companion files, such as tests, documentation, or registries, known only from repository history, which the skills supply. However, File IoU displays only a weak positive correlation with pass rates at the task level (point-biserial

$r = 0.12$ for Claude Opus 4.6). Modifying the correct files is a necessary condition for passing tests rather than a sufficient one, affirming our positioning of File IoU as an auxiliary diagnostic of localisation rather than a correctness proxy. Nevertheless, in repositories enforcing strict conventions, an organicity failure is literally a test failure, a dynamic illustrated by the qutebrowser case study in Appendix H. Panel (b) reveals the computational mechanics behind the overall gain: skill-conditioned agents consume markedly more tool calls on *helped* task-runs. As detailed in Appendix H, this reflects deliberate, repository-aware exploration where agents leverage historical commit conventions to unlock previously unreachable solutions. Conversely, the regressions highlight a recurring *tunnel effect*: loaded skills can occasionally prime the agent so confidently that it skips the exploratory greps needed to safely ground generic priors. Finally, Table 3 demonstrates a clear scaling trend: the absolute performance lift grows monotonically across Kimi K2.5, Claude Sonnet 4.6, and Claude Opus 4.6 ($+3.0 \rightarrow +6.0 \rightarrow +8.0$ pp), suggesting that stronger base models exhibit higher quality in both accumulating and reusing experience. Appendix I.5 complements this analysis with compute-matched baselines and per-task token accounting.

5 Conclusion and Future Work

We presented **Learning to Commit**, an agentic framework that actively learns from a repository's evolution: by performing on-policy contrastive reflection on historical commits, the agent distils implicit architectural constraints and coding conventions into a reusable skill memory. On the Learning to Commit benchmark this memory significantly improves file localisation and reduces patch bloat, and on SWE-bench Pro it further lifts test-pass rates, confirming that organicity translates into functional correctness. A natural next step is to turn the contrastive blind-attempt-versus-oracle signal into a reinforcement learning objective and train the agent end-to-end to acquire repository-specific development experience, moving from skill distillation to skill optimisation.

Acknowledgments and Disclosure of Funding

We thank L.H. Xu for valuable discussions and suggestions in the early stage of this work. This work was supported in part by Tsinghua University (AIR)-AsiaInfo Technologies (China), Inc. Joint Research Center for 6G Network and Intelligent Computing (20233910006), and Wuxi Research Institute of Applied Technologies, Tsinghua University under Grant 20242001120.

References

[1] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

[4] Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. Fea-bench: A benchmark for evaluating repository-

level code generation for feature implementation, 2025. URL https://arxiv.org/abs/2503.06680.

[5] Qixing Zhou, Jiacheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, Dandan Tu, and Zhaoxiang Zhang. Featurebench: Benchmarking agentic coding for complex feature development, 2026. URL https://arxiv.org/abs/2602.10975.

[6] Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library generation from scratch, 2024. URL https://arxiv.org/abs/2412.01769.

[7] Jialong Chen, Xander Xu, Hu Wei, Chuan Chen, and Bing Zhao. Swe-ci: Evaluating agent capabilities in maintaining codebases via continuous integration, 2026. URL https://arxiv.org/abs/2603.03823.

[8] Boshi Wang, Weijian Xu, Yunsheng Li, Mei Gao, Yujia Xie, Huan Sun, and Dongdong Chen. Improving code localization with repository memory, 2026. URL https://arxiv.org/abs/2510.01003.

[9] Quanjun Zhang, Chengyu Gao, Yu Han, Ye Shang, Chunrong Fang, Zhenyu Chen, and Liang Xiao. Sgagent: Suggestion-guided llm-based multi-agent framework for repository-level software repair, 2026. URL https://arxiv.org/abs/2602.23647.

[10] Jiayi Wei, Greg Durrett, and Isil Dillig. Coeditor: Leveraging contextual changes for multi-round code auto-editing, 2024. URL https://arxiv.org/abs/2305.18584.

[11] Qinglin Zhu, Tianyu Chen, Shuai Lu, Lei Ji, Runcong Zhao, Murong Ma, Xiangxiang Dai, Yulan He, Lin Gui, Peng cheng, and Yeyun Gong. Pull requests as a training signal for repo-level code editing, 2026. URL https://arxiv.org/abs/2602.07457.

[12] Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents, 2025. URL https://arxiv.org/abs/2504.07164.

[13] Thomas Joshi, Shayan Chowdhury, and Fatih Uysal. Swe-bench-cl: Continual learning for coding agents, 2025. URL https://arxiv.org/abs/2507.00014.

[14] Huichi Zhou, Yihang Chen, Siyuan Guo, Xue Yan, Kin Hei Lee, Zihan Wang, Ka Yiu Lee, Guchun Zhang, Kun Shao, Linyi Yang, and Jun Wang. Memento: Fine-tuning llm agents without fine-tuning llms, 2025. URL https://arxiv.org/abs/2508.16153.

[15] Tianxin Wei, Noveen Sachdeva, Benjamin Coleman, Zhankui He, Yuanchen Bei, Xuying Ning, Mengting Ai, Yunzhe Li, Jingrui He, Ed H. Chi, Chi Wang, Shuo Chen, Fernando Pereira, Wang-Cheng Kang, and Derek Zhiyuan Cheng. Evo-memory: Benchmarking llm agent test-time learning with self-evolving memory, 2025. URL https://arxiv.org/abs/2511.20857.

[16] Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of llm agents, 2024. URL https://arxiv.org/abs/2402.17753.

[17] Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. Longmemeval: Benchmarking chat assistants on long-term interactive memory, 2025. URL https://arxiv.org/abs/2410.10813.

[18] Lilin Wang, Lucas Ramalho, Alan Celestino, Phuc Anthony Pham, Yu Liu, Umang Kumar Sinha, Andres Portillo, Onassis Osunwa, and Gabriel Maduekwe. Swe-bench++: A framework for the scalable generation of software engineering benchmarks from open-source repositories, 2025. URL https://arxiv.org/abs/2512.17419.

[19] Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks?, 2025. URL https://arxiv.org/abs/2509.16941.

[20] Jiale Zhao, Guoxin Chen, Fanzhe Meng, Minghao Li, Jie Chen, Hui Xu, Yongshuai Sun, Wayne Xin Zhao, Ruihua Song, Yuan Zhang, Peng Wang, Cheng Chen, Jirong Wen, and Kai Jia. Immersion in the github universe: Scaling coding agents to mastery, 2026. URL https://arxiv.org/abs/2602.09892.

[21] Nikita Benkovich and Vitalii Valkov. Agyn: A multi-agent system for team-based autonomous software engineering, 2026. URL https://arxiv.org/abs/2602.01465.

[22] Shangqing Liu, Yanzhou Li, Xiaofei Xie, and Yang Liu. Commitbart: A large pre-trained model for github commits, 2023. URL https://arxiv.org/abs/2208.08100.

[23] Jared Zhu, Minhao Hu, and Junde Wu. Swe context bench: A benchmark for context learning in coding, 2026. URL https://arxiv.org/abs/2602.08316.

[24] Pengfei Du. Memory for autonomous llm agents:mechanisms, evaluation, and emerging frontiers, 2026. URL https://arxiv.org/abs/2603.07670.

[25] Kangning Shen, Jingyuan Zhang, Chenxi Sun, Wencong Zeng, and Yang Yue. Structurally aligned subtask-level memory for software engineering agents, 2026. URL https://arxiv.org/abs/2602.21611.

[26] Yaolun Zhang, Yiran Wu, Yijiong Yu, Qingyun Wu, and Huazheng Wang. Live-evo: Online evolution of agentic memory from continuous feedback, 2026. URL https://arxiv.org/abs/2602.02369.

[27] Yinjie Wang, Xuyang Chen, Xiaolong Jin, Mengdi Wang, and Ling Yang. Openclaw-rl: Train any agent simply by talking, 2026. URL https://arxiv.org/abs/2603.10165.

[28] Ivan Yee Lee, Loris D'Antoni, and Taylor Berg-Kirkpatrick. The format tax, 2026. URL https://arxiv.org/abs/2604.03616.

[29] Anthropic. Introducing claude opus 4.6. https://www.anthropic.com/news/claude-opus-4-6, feb 2026. Official Anthropic model announcement.

[30] Google. Gemini 3 pro. https://blog.google/innovation-and-ai/technology/developers-tools/gemini-3-pro-vision/, 2025. Official Google announcement.

[31] OpenAI Team. Gpt-4o system card, 2024. URL https://arxiv.org/abs/2410.21276.

A Limitations

While *Learning to Commit* demonstrates the efficacy of online repository memory, several aspects of the current framework provide fertile ground for future extension. First and foremost, our current **skill induction mechanism** relies exclusively on code changes within the commit history. In a professional onboarding setting, a developer's mental model is also shaped by "off-chain" knowledge—design documents, peer review nuances in PR threads, and informal architectural discussions in Slack or Discord. Integrating these heterogeneous and often conflicting sources into a unified, conflict-aware skill memory is a non-trivial challenge that this work does not yet address. A second consideration is **domain-specific generalization**. Our empirical evaluation primarily focuses on AI-engineering repositories characterized by rapid iteration and specific coding paradigms (e.g., LLM training and vector databases). It remains to be seen how the framework adapts to domains with vastly different development cultures, such as the rigorous memory safety requirements of OS kernels, the performance-critical latency of game engines, or the complex business logic of financial back-ends.

A third limitation lies in the **robustness of the learned skills and the quality of commit histories**. An audit of 60 sampled skill entries (Appendix I.6) reveals a 5% hard factual error rate. A more insidious failure mode occurs when an entry is factually correct but overly specialised to its source commit. Such rigid rules are not strictly wrong but lower the reusability of the skill memory. Furthermore, our framework relies on the quality of the underlying history. The quality filter discussed in §3.4 drops trivial changes before learning. A noisier history therefore results in fewer usable experiences rather than strictly dirtying the memory. The consistent improvements on the 10 community projects in the SWE-bench Pro evaluation indicate that textbook-quality histories are not required. However, for extremely noisy repositories, the onboarding material would eventually deplete.

Finally, our evaluation methodology involves **relative metrics and benchmark construction constraints**. The deterministic metrics use the oracle commit as a reference. This approach is naturally conservative against valid alternative implementations. Human evaluation (Appendix D) shows that 10% of the sampled patches are acceptable alternatives, a proportion that doubles below a File IoU of 0.5. Low File IoU therefore mixes genuine localisation failures with legitimate implementation divergences. Additionally, the held-out test sets are relatively small, comprising 50 tasks for the main benchmark and 50 tasks for the SWE-bench Pro subset. This size is a deliberate choice under compute constraints (full project API costs are detailed in Appendix I.7). The automated construction pipeline can mechanically scale to more repositories and later time windows. The pipeline also uses LLMs extensively for quality filtering, clustering, and synthetic query generation, which introduces the possibility of construction bias. A dual audit of 30 sampled queries showed 0/30 hard leaks and a mean realism score of 4.0/5. However, this finite audit cannot fully rule out construction bias.

B Broader Impacts

Positive impacts. The framework can lower the pull-request review burden on open-source maintainers by producing patches that align with a repository's own idioms; this is especially useful for smaller teams without dedicated review capacity, and the distilled skill memory can additionally serve as on-ramp documentation for newcomers.

Negative impacts. Two risks merit attention. First, by lowering the cost of producing organic-looking code, the framework may accelerate the replacement of human maintainers and erode the transmission of tacit knowledge that does not appear in commit history. Second, because skill induction reads from raw commit history, a repository whose history has been deliberately seeded with malicious patterns (subtle backdoors, insecure defaults) could propagate those patterns through the skill memory into future patches; oracle-side filtering of suspicious commits or human review of the distilled skill document before deployment can mitigate this risk, but neither is implemented in this paper.

C Saturation of Existing Long-Term Memory Benchmarks

To support the claim in §1 that existing long-term memory benchmarks are largely saturated by frontier models, we evaluate Gemini 3 Pro [30] and GPT-4o [31] on LongMemEval [17]—a widely used long-term conversational memory benchmark—under the *with-context* setting, where the entire conversational history is concatenated and provided as a single prompt. Results are summarised in Table 4.

Gemini 3 Pro reaches 90% or higher on every per-split metric except single-session-preference (83.3%), with the largest absolute gains over GPT-4o on multi-session (+38.2 pp), single-session-preference (+50.0 pp), and temporal-reasoning (+21.0 pp). **Upon manual inspection, we find that most residual failure cases are not genuine memory errors but stem from underspecified or borderline ground-truth annotations**. For example, when asked "how many projects have I led", the oracle excludes projects the user merely intended to start, while the model counts them; when asked which clothing items were considered before a purchase, the oracle treats briefly mentioned items as already-rejected, while the model treats them as still under consideration. Under the with-context setting, the residual headroom on LongMemEval is therefore dominated by annotation am-

Table 4: Accuracy on LongMemEval under the with-context setting. Gemini 3 Pro substantially outperforms GPT-4o on every split, reaching 93.6% overall.

Split	Gemini 3 Pro	GPT-4o
Overall	**93.6%**	71.6%
single-session-assistant	100.0%	100.0%
single-session-user	100.0%	94.3%
temporal-reasoning	93.2%	72.2%
knowledge-update	93.6%	78.2%
multi-session	90.1%	51.9%
single-session-preference	83.3%	33.3%

biguity rather than memory capability, supporting our claim that conversational-recall benchmarks no longer offer meaningful difficulty for evaluating long-term memory in frontier models.

D Human Studies

D.1 LLM-Judge Validation against Human Pairwise Ratings

To verify that our pairwise LLM-judge protocol reliably tracks human preferences, we collect expert pairwise ratings on a held-out sub-sample drawn from an internal LLM RL training repository where in-house annotators provide ground-truth verdicts, and compare them against the LLM-judge results along the four organicity dimensions of scope alignment, logic similarity, redundancy & hallucination, and code style. The validated protocol is then applied to evaluate organicity on the Learning to Commit benchmark (§4.2). Table 5 reports per-dimension agreement between human raters and the LLM judges.

Table 5: Agreement between human pairwise ratings and the LLM judges across organicity dimensions, on an expert-rated sub-sample of 100 pairs (the same pairs are rated along each of the four dimensions). We report exact-match agreement (p_o) and a uniform-chance-corrected agreement score, computed as $(p_o - 1/3)/(1 - 1/3)$, where $1/3$ is the chance baseline under random three-way choice over `A_WIN` / `B_WIN` / `TIE`. The Overall row is the macro-average across the four dimensions.

Dimension	Human–LLM Agreement (%)	Chance-corrected Agreement
Scope Alignment	74	0.61
Logic Similarity	70	0.55
Redundancy & Hallucination	81	0.72
Code Style	74	0.61
Overall	**75**	**0.62**

D.2 Human Mergeability Study

Pairwise preference is not the ultimate metric engineers care about. The actual question is whether a maintainer would merge the generated patch. To upgrade our validation to direct merge verdicts, we conducted a dual-track blind evaluation study.

For Track A, we focus on real maintainers evaluating an internal repository. We chronologically selected 30 recent merge requests from the history of an internal LLM reinforcement learning repository. We programmatically filtered out trivial changes such as pure configuration edits, version bumps, or patches with fewer than 10 lines. The baseline agent and the skill-conditioned agent each generated one patch for every task description. Two maintainers of the repository independently and blindly evaluated all 60 patches. The patches were presented anonymously in a random order without condition labels. Raters assigned one of three verdicts: approve, request changes, or reject. The criteria asked how they would handle the patch as a maintainer. We used an any-approve rule for aggregation. A patch received an approve if any maintainer approved it. Otherwise, it received a request changes if any maintainer requested changes. It received a reject only if both maintainers rejected it. Regarding conflicts of interest, Maintainer 1 is an author of this paper and an active

maintainer of the repository, and Maintainer 2 is a non-author maintainer. The standalone validity holds even when looking only at the non-author maintainer, who approved 17 out of 30 patches for the skill condition against 6 out of 30 for the baseline.

As summarised in Table 6, the skill condition secured 21 approves, 7 request changes, and 2 rejects across the 30 tasks. This represents a substantial improvement against the baseline agent, which received 8 approves, 16 request changes, and 6 rejects.

Table 6: Track A: merge verdicts from two real maintainers on 30 internal GitLab merge-request tasks, aggregated per MR under the any-approve rule. Per-maintainer verdicts are released separately in the supplementary material.

Condition	Approve	Request changes	Reject
Baseline agent (no skill)	8/30	16/30	6/30
Skill-conditioned agent (ours)	**21/30**	7/30	**2/30**

For Track B, we evaluate the public benchmark repositories with external annotators. We sampled 10 held-out tasks from each of three main benchmark repositories (vLLM, Milvus, DeepSpeed) to form a pool of 30 tasks. Each task was paired with two anonymous patches corresponding to the baseline and skill conditions in a randomised order. Three annotators with maintainer-level industrial experience independently reviewed all 60 patches. We invited these annotators through the professional network of the authors. Each annotator has over three years of experience maintaining large codebases and is not an author of this paper. They used the same three-tier verdict system. They additionally marked whether each patch constituted an acceptable alternative implementation that deviates from the oracle. Both tracks maintained a strict blind scope. Annotators knew they were reviewing patches produced by an automated coding system, but they did not know which condition produced which patch, nor were they informed of any system details or research hypotheses.

Table 7 details the Track B results based on 90 votes per condition from 3 annotators across 30 patches. The skill condition achieved 48% approve, 40% request changes, and 12% reject. It outperformed the baseline condition which recorded 32% approve, 44% request changes, and 23% reject. The inter-annotator agreement yielded a moderate Fleiss' κ of 0.52.

Table 7: Track B: merge verdicts from three maintainer-level annotators on 30 benchmark tasks (60 anonymised patches). Percentages count all 90 verdicts per condition (3 annotators $\times$ 30 patches); rows may not sum to 100% due to rounding.

Condition	Approve	Request changes	Reject
Baseline agent (no skill)	32%	44%	23%
Skill-conditioned agent (ours)	**48%**	40%	**12%**

We associate our proxy metrics with these human merge verdicts in Table 8. We encoded verdicts as 0 for reject, 1 for request changes, and 2 for approve, and computed the mean across the three raters. File IoU achieves a patch-level Spearman correlation of 0.40 with human merge verdicts. Line deviation achieves a correlation of -0.28. The pairwise judge winner agrees with the better human verdict on 21 out of 30 tasks, corresponding to a 70% agreement rate.

Table 8: Association between each organicity proxy and the Track-B human merge verdicts (60 patches). Verdicts are coded reject $= 0$, request changes $= 1$, approve $= 2$ and averaged across annotators. For the judge row, a task counts as agreeing when the judge's pairwise winner matches the patch with the better human verdict.

Proxy	Association with human merge verdict
File IoU	Spearman $\rho = 0.40$ (patch level)
Line deviation	Spearman $\rho = -0.28$ (patch level)
Pairwise judge (task-level overall verdict)	agrees on 21/30 tasks (70%)

The evaluation of acceptable alternatives quantifies legitimate deviations from the oracle. Out of all 60 patches, 6 patches representing 10% of the total were marked as acceptable alternatives by

at least two-thirds of the annotators. Among the 15 patches with a File IoU below 0.5, this proportion rises to 3 patches or 20%, which is double the overall rate. This indicates that low scores on deterministic metrics encompass a fraction of legitimate implementation divergences. It confirms our methodology of using File IoU strictly as an auxiliary diagnostic for localisation rather than a definitive correctness metric.

We have released comprehensive materials in the supplementary files. These include all 60 patch files for Track B containing task descriptions, oracle diffs, and the anonymous patches. We also provide the per-patch verdict tables from the three annotators and the full annotation guidelines. The anonymous per-MR verdict table and a one-page review protocol for Track A are similarly included, omitting only the proprietary MR content and code of the internal repository.

E Learning to Commit Benchmark Details

Selection criteria. We select repositories satisfying: (i) AI-domain relevance—training, inference, vector retrieval, modelling, or distributed AI; (ii) rigorous PR review with consistent commit-message conventions; (iii) substantive engineering complexity beyond glue code; (iv) ongoing active maintenance; and (v) multi-language coverage. The resulting 5 repositories collectively span 63,417 non-merge commits over 2.6–7.1 years of project history.

Table 9: Repository metadata for the Learning to Commit benchmark. Star counts are rounded to the nearest thousand at the time of construction; # Commits and Span are computed from each repository's full non-merge history. Sub-domains cover inference, training optimisation, and vector retrieval.

Repository	Sub-domain	Language	Stars	# Commits	Span (yr)
vllm-project/vllm	LLM inference	Python	75K	16,258	3.2
sgl-project/sglang	LLM inference	Python	25K	12,191	2.6
microsoft/DeepSpeed	Training optimisation	Python	36K	3,136	6.3
ggml-org/llama.cpp	LLM inference	C/C++	100K	8,998	3.1
milvus-io/milvus	Vector database	Go (+C++)	32K	22,834	7.1
Total / range	—	—	—	63,417	2.6–7.1

Per-dimension breakdown of skill gains. Table 10 reports the per-dimension pairwise judgment outcomes of the skill-conditioned agent (Claude Opus 4.6) against the vanilla baseline, broken down across the four learning configurations of §4.2. Each cell shows the percentage of judgments in which the skill agent wins versus the vanilla agent wins; the residual is ties. The pattern is consistent with the dimensional attribution reported in §4.2: Q2 (logic similarity) and Q3 (redundancy and hallucination) show the strongest preference for the skill-conditioned agent, with Q3 favouring skill in three of the four configurations, while Q1 (scope alignment) and Q4 (code style) are largely neutral. The `seq-bycat` configuration is the only one in which the vanilla agent wins more often than skill across all four dimensions, consistent with its overall pairwise win rate of 42% in Table 2.

Table 10: Per-dimension pairwise judgments (skill-conditioned agent vs. vanilla, Claude Opus 4.6) across the four learning modes. Each cell shows skill-wins / vanilla-wins as percentages; ties are the residual. Bold cells mark dimensions where skill leads vanilla by ≥ 15 percentage points.

Configuration	Q1 Scope	Q2 Logic	Q3 Redund.	Q4 Style
`seq-all` (ref)	36.5 / 30.0	**49.5 / 23.5**	**53.0 / 21.0**	33.5 / 29.0
`par-bycat`	36.0 / 28.5	**46.0 / 24.5**	**47.5 / 22.0**	33.0 / 28.0
`par-all`	33.0 / 30.0	39.5 / 29.0	39.5 / 27.5	30.0 / 30.5
`seq-bycat`	24.5 / 39.0	23.0 / 45.0	27.0 / 40.5	22.5 / 38.5

Extended code-review dimensions. We rerun the pairwise judge on existing patch pairs across four new code review dimensions with 200 rounds per dimension, swapping the presentation order of the two patches to eliminate position bias and recording unable-to-judge rounds as ties. Table 11 details the results. The skill condition leads across all four new dimensions, with maintainability and

regression risk showing the highest direct win rates, architectural fit also favouring the skill agent, and test adequacy producing the most ties. Broader criteria covering API design, performance, security, and long-term technical debt are covered by the human merge study where annotators weigh all these factors simultaneously against the task requirements, as detailed in Appendix D.

Table 11: Pairwise judge results on four additional code-review dimensions (skill-conditioned agent vs. vanilla, Claude Opus 4.6, 200 judging rounds per dimension with presentation order alternated across rounds; undecided rounds recorded as ties).

New dimension	Skill wins	Vanilla wins	Ties
Maintainability	**44.0%**	31.5%	24.5%
Test adequacy	**21.5%**	16.0%	62.5%
Architectural fit	**32.5%**	18.0%	49.5%
Regression risk	**41.0%**	27.5%	31.5%

Representative case studies (milvus). We highlight two tasks from milvus's held-out test split on which we observed clear, mechanistic organicity gains under skill conditioning. **Lock-free segment refactor.** The task requires the agent to extend a concurrent segment manager with a lock-free fast path. The no-skill baseline edits an unrelated API surface entirely (File IoU 0%), whereas the skill-conditioned agent localises directly to the segment-manager file (File IoU 83%) and wins all four ABBA pairwise judgments (4-0). The milvus skill document had recorded a project-wide convention that long-running data structures register through a writer-priority shared-mutex helper rather than ad-hoc locking—exactly the convention the test commit relies on. **Compaction self-heal.** The task requires the agent to fix a state-machine corner case in compaction triggered by a coordinator restart. Both agents identify the right module, but the no-skill baseline misses the project's standard self-heal idiom (re-emit the in-flight plan on coordinator-restart, log at WARN, continue) and instead reaches for a one-shot retry. The skill-conditioned agent applies the in-house pattern, lifting File IoU from 38% to 57% and again sweeping the pairwise comparison 4-0.

F Adapting SWE-bench Pro to the Temporal-Split Protocol

This appendix describes how we adapt the public SWE-bench Pro benchmark [19] to the per-repository temporal-split protocol of §3.4. We make no changes to the original test tasks themselves—only an additional onboarding step that supplies historical learning commits before each test task, so any pass@1 delta we report is directly attributable to the onboarding phase.

Source dataset. We start from the official SWE-bench Pro release on HuggingFace (`ScaleAI/SWE-bench_Pro`), which provides **731 tasks across 11 repositories**. Each task ships with: (i) a problem statement; (ii) a `base_commit` the task is rooted at; (iii) the oracle patch (git diff) that solves the task; (iv) two test sets, `fail_to_pass` and `pass_to_pass`, that together define correctness; and (v) a Docker image with the project's runtime environment pre-installed. Resolution is judged by applying a candidate patch inside the container and checking whether $(\texttt{fail_to_pass} \cup \texttt{pass_to_pass}) \subseteq$ passed_tests.

Subsampling. For tractability, we perform stratified sampling with near-uniform quotas per repository across 10 of the 11 original repositories (`webclients` excluded) to select 50 tasks from the 731-task corpus. Table 12 reports the per-repository task counts in the subsample alongside the original SWE-bench Pro counts and primary languages. By language, the 50 sampled tasks comprise 17 Python, 21 Go, 8 JavaScript, and 4 TypeScript tasks, preserving SWE-bench Pro's language coverage.

Construction pipeline. The full adaptation runs in four stages.

(s0) Image mirroring. We mirror the official SWE-bench Pro Docker images into our evaluation environment, and the sandbox loads the per-task image directly at evaluation time.

(s1) Learning-commit selection. For each test task with `base_commit` c, we (i) parse the repository's `git log` for non-merge, non-bot, non-trivial commits with date strictly earlier than c (excluding c itself), (ii) apply the same programmatic pre-filters used in §3.4 stage (i), and (iii) pass the

Table 12: Per-repository task counts in our SWE-bench Pro 50 subsample versus the original SWE-bench Pro corpus.

Repository	Language	Original # Tasks	Our Subsample
ansible	Python	96	6
openlibrary	Python	91	6
flipt	Go	85	6
qutebrowser	Python	79	5
teleport	Go	76	6
vuls	Go	62	5
navidrome	Go	57	4
element-web	JavaScript	56	4
NodeBB	JavaScript	44	4
tutanota	TypeScript	20	4
Total	—	666	50

surviving candidates through the two-stage decoding scheme (Gemini for free-form scoring, Claude for structured extraction; cf. *format tax* [28]) to select the top-3 most relevant historical commits. The selected commits are sorted oldest-first to give the agent a chronologically faithful learning trajectory and to avoid the knowledge-regression failure mode discussed in Appendix H.

(s2) Benchmark assembly. For each selected learning commit we materialise (a) its full diff, (b) a `git archive` snapshot of the repository at the commit, and (c) a synthesised issue-style natural-language query generated from the commit message and diff under the same prompt template used for the Learning to Commit benchmark. We then assemble each test task into a unified record carrying the SWE-bench Pro problem statement, oracle patch, test specification, base snapshot, and the ordered list of 3 learning artifacts. A short environment hint is appended to the problem statement informing the agent that the sandbox lacks the full runtime (the official Docker image is reserved for evaluation, not for solve-time exploration). Output: 50 test records and 150 learning artifacts in total.

(s3) Sandbox evaluation. At evaluation time we boot the official SWE-bench Pro Docker image for the task, apply the candidate patch, run the official test runner, and parse `fail_to_pass`/`pass_to_pass` status. The harness supports concurrent evaluation across multiple solve directories (with-skill vs. baseline) and incremental writes for resumable runs. Pass@1 in Table 3 is the fraction of tasks meeting the official resolution criterion. To control for run-to-run agent stochasticity, we repeat the entire solve-and-evaluate procedure 4 times independently for each (model, condition) cell, yielding 200 task-runs per cell; reported pass@1 is the mean across the 4 rollouts and reported Δ is the per-run paired difference (skill minus baseline) averaged across the 4 rollouts, with their standard deviations computed across the same 4 rollouts.

Summary of the difference. Official protocol: *problem* $\rightarrow$ *agent solves* $\rightarrow$ *Docker test verdict*. Ours: *learning commits* $\rightarrow$ *agent onboards* $\rightarrow$ *problem* $\rightarrow$ *skill-conditioned solve* $\rightarrow$ *same Docker test verdict*. Test tasks and grading are unchanged.

G Contamination Audit

Evaluations using public repositories frequently encounter concerns regarding pre-training data contamination. Specifically, one might ask if the high File IoU scores stem from models simply recalling the correct files from their pre-training data rather than genuinely improving through skill conditioning. We conduct a contamination audit across four dimensions to address this.

The first dimension involves a paired difference argument. Every performance gain reported in our main results represents a difference within the same base model. Any memorised repository knowledge is equally available to both the baseline and the skill-conditioned configurations. If the File IoU gains were driven by memorisation, the baseline agent would benefit equally and the difference would vanish. We observe consistent differences from 5 to 7 percentage points across all three tested models. Figure 2 further demonstrates that the held-out File IoU climbs steadily across onboarding iterations. It rises from 60% to 65% for vllm and from 56% to 67% for milvus. The

base model weights remain entirely frozen during this process, confirming that the improvements arise from the accumulated skill memory rather than pre-trained weights.

The second dimension is a temporal audit. We examined the commit dates of the evaluation tasks relative to model knowledge cutoffs. 45 out of the 50 test commits are dated in or after February 2026, which is the release month of Claude Opus 4.6, the latest model evaluated in our study. Four out of the five repositories have all 40 out of 40 test commits satisfying this strict post-release criterion. The 5 exceptions are all located in DeepSpeed, dating from November 2025 to January 2026. When we exclude these 5 tasks, the File IoU difference remains at +5.0 percentage points. This matches the +5.0 percentage points observed on the full suite of five repositories.

The third dimension employs a snapshot-free memory probe. We provided the model solely with the synthetic issue text, which strictly omits file paths and function names based on step (v) of our data pipeline, and requested it to guess the modified files without any repository access. Relying strictly on memory, Claude Opus 4.6 scores a File IoU of 0.15 and Claude Sonnet 4.6 scores 0.16. This represents roughly one quarter of the 0.53 to 0.67 range achieved by agents interacting with the repository as shown in Table 2. The probe scores for both models are nearly identical despite their notably different skill gains.

The fourth dimension conducts an audit on the synthetic query construction. We double-audited a sample of 30 queries for potential leakage and realism. On the leakage side, zero queries contained verbatim oracle file paths, internal symbols, or code-level solution steps, resulting in a hard leakage count of 0 out of 30. A string-matching scan flagged 9 loose hits involving public feature names. All 30 queries fall into a middle band in granularity, being neither verbatim leaks nor completely under-specified. This is an inherent property of reverse-synthesising tasks from merged commits. On the realism side, auditors rated the queries with an average score of 4.0 out of 5 when compared against authentic issue reports. The auditors explicitly noted that the queries read like task tickets written by maintainers.

H Case Studies: Where Skills Help and Where They Hurt

Internal validation runs: where skills help and occasionally over-caution. On runs from our internal expert-maintained LLM RL training repository (used for LLM-judge calibration; see §D), the largest positive effect of skill conditioning appears on a task requiring the agent to fix a shared-RNG concurrency bug in a judge client module. The skill agent, having learned the repository's module structure during onboarding, correctly localises the fix to the judge client file (File IoU = 100%), while the baseline agent misidentifies the target entirely, editing an unrelated API module (File IoU = 0%). Conversely, the largest negative effect occurs on a training-step guard logic fix, where both agents correctly identify the core bug but the skill agent retains a redundant defensive guard that the baseline omits, resulting in slightly more verbose code. This illustrates that skills occasionally induce over-caution without functionally harming the solution.

SWE-bench Pro `f8e7fea0` **(qutebrowser): dynamic settings via** `Callable`**.** The task asks for a tri-state setting `qt.workarounds.disable_accelerated_2d_canvas` (`always`/`never`/`auto`) where the `auto` branch disables the accelerated 2D canvas only on Qt 6 with Chromium below 111. The baseline agent passes 87 of 88 fail-to-pass tests (and 20/20 pass-to-pass, but the task is reported as *unresolved*): the runtime-version branching is logically correct, but it is implemented as an ad-hoc `if/elif` patched onto the `qtargs` assembly site, which (i) bypasses the central `_WEBENGINE_SETTINGS` registry that the rest of the file uses, and (ii) omits the required entry in `doc/help/settings.asciidoc`. Consequently, the dictionary-driven settings-existence test fails, while the asciidoc omission stands as a silent deviation from repository conventions without triggering a test failure. The skill-conditioned agent reaches 88/88 (resolved) by following three repository-specific conventions that its onboarding has distilled and that generic LLM priors cannot supply: (a) settings whose *purpose* is to work around an upstream rendering bug belong under the `qt.workarounds.*` namespace, not `qt.chromium.*`, even though both ultimately emit Chromium flags; (b) `_WEBENGINE_SETTINGS` is the canonical extension point for new flags, and dynamic settings are registered by writing a `Callable` *that returns another key in the same dict*—e.g., `lambda versions, ns, sf: 'always' if machinery.IS_QT6 and versions.chromium_major < 111 else 'never'`—so runtime detection is funnelled through validated indirection rather than raw flag strings; (c) despite the prominent "DO NOT EDIT" dis-

Baseline agent 87/88, unresolved ✗	**Skill-conditioned agent** 88/88, resolved ✓
1. ✓ Locate `configdata.yml` and `qtargs.py`; identify `experimental_web_platform_features` as a tri-state analogue.	1. ✓ Read `SKILL.md` + `config_and_qtargs.md` + `testing.md` before exploring code.
2. ✓ Add new entry under `qt.workarounds.*` with type, default, backend.	2. ✓ Confirm `qt.workarounds.*` namespace per the skill's "purpose is to work around a bug" rule.
3. ✗ Inspect `_WEBENGINE_SETTINGS`; conclude its `auto` branch is resolved at *import time* and so *cannot* carry runtime state.	3. ✓ Add new entry to `configdata.yml` (same as baseline).
4. ✗ Patch `_qtwebengine_args` directly with an inline `if/elif` on `disable_canvas`, bypassing the central registry.	4. ✓ Recognise `_WEBENGINE_SETTINGS` as canonical extension point; dynamic values register as a `Callable` returning another key.
5. ✓ Add parametrised tests covering `always/never/auto`.	5. ✓ Extend the registry: `lambda versions, ns, sf: 'always' if IS_QT6 and chromium_major < 111 else 'never'`; widen `_qtwebengine_settings_args` to dispatch callables.
6. ✗ Skip `settings.asciidoc`, deferring to its "DO NOT EDIT" banner.	6. ✓ Edit `settings.asciidoc` (summary table + detail section) per repo's commit convention.
7. ✗ Submit. Dictionary-driven existence test and doc-consistency test fail.	7. ✓ Add tests using the `version_patcher` + `monkeypatch IS_QT6` pattern from the skill.

Figure 4: Step-level trajectory comparison on SWE-bench Pro task `qutebrowser__qutebrowser-f8e7fea0`. Both agents share the same Claude Opus 4.6 backbone, the same tool set, and the same user prompt; the only difference is whether the skill document distilled during onboarding is provided. The decisive divergence is at step 4: the baseline reads `_WEBENGINE_SETTINGS` but rejects it as the extension point because its `auto` arm is resolved at import time, and falls back to ad-hoc `if/elif` branching at the assembly site; the skill-conditioned agent knows from prior commits that the registry's contract has been incrementally widened to accept a `Callable` returning another key in the same dict, and reuses that indirection. The downstream omission of `settings.asciidoc` (step 6) similarly reflects which conventions are visible in the commit history but invisible to a fresh reader of the file.

claimer at the top of `settings.asciidoc`, every config-touching commit in this repository hand-edits the file, and so must this one. The case crystallises the central thesis: pre-training priors reproduce the surface logic, but the load-bearing decisions—*which extension point to use* and *how the project has incrementally taught that extension point new tricks*—are recoverable only from the commit history. Figure 4 contrasts the two trajectories step by step: identical surface logic, divergent extension-point choice.

SWE-bench Pro `f343c08f` (openlibrary): mocks are project code, not immutable infrastructure. The task fixes the catalog's author-matching layer: `remove_author_honorifics` must change from `dict` → `dict` to `str` → `str` and tolerate honorific-only inputs (`"Mr."`); `find_author`'s queries must escape `*` in author names and use four-digit-year wildcards (`*1829*`) for the surname branch so that `"September 14th, 1829"` and `"1829-09-14"` match. The baseline agent produces an empty patch (not even submitted to the evaluator): it reaches the same diagnosis but, on noticing that escaping `*` as `\*` interacts badly with the test mock's `regex_ilike`—which naively rewrites `*` to `.*`—enters a deliberation spiral over query-side-only workarounds (`replace('*','')`, `[*]`, `re.escape`, no escape). Its final long-form deliberation hits the 16K single-response token limit and is truncated mid-sentence. The agent loop then terminates because the truncated response contains no tool calls, without a single `edit_file` ever being issued. The skill-conditioned agent reaches 2/2 fail-to-pass and 7/7 pass-to-pass (resolved) by acting on three repository-specific facts that its onboarding has distilled and that generic LLM priors do not supply: (a) `openlibrary/mocks/mock_infobase.py` is in-tree project code and is the

canonical fix site when the mock under-approximates production `ILIKE`—the skill's `SKILL.md` explicitly enumerates it as "Mock infrastructure: `openlibrary/mocks/mock_infobase.py` (MockSite), with tests in `openlibrary/mocks/tests/`", framing it as code to fix rather than a constraint to work around; (b) the author-processing pipeline runs `remove_author_honorifics` *before* `east_in_by_statement` inside `build_query`, so the call site must be rewired as `author['name'] = remove_author_honorifics(author['name'])` rather than passing the dict through; (c) `HONORIFC_NAME_EXECPTIONS` (note the in-repo typo) is a `frozenset` of case-folded keys, not a dict, and the honorific-only guard must `return name` unchanged after the strip, not the empty residue. With (a) unblocking the implementation, the agent extends `regex_ilike` with a placeholder technique (`'\x00STAR\x00'`) that round-trips backslash-escaped `\*` through the `*` → `.*` rewrite as a literal star, then escapes `*` symmetrically in the three `find_author` queries.

Baseline agent empty patch ✗	**Skill-conditioned agent** 2/2 f2p, 7/7 p2p, resolved ✓
1. ✗ Read `load_book.py`, `helpers.py`, `test_load_book.py`; enumerate the four sub-fixes.	1. ✓ Read `SKILL.md` + `catalog_patterns.md` + `testing.md` before opening any source file.
2. ✓ Identify the dict-to-`frozenset` conversion of `HONORIFC_NAME_EXECPTIONS` and the `str` → `str` rewrite of `remove_author_honorifics`.	2. ✓ Convert `HONORIFC_NAME_EXECPTIONS` to a `frozenset`; rewrite `remove_author_honorifics(name: str) -> str` with the honorific-only guard.
3. ✗ Realise that escaping `*` as `\*` in name queries collides with the mock's `regex_ilike`, which naively does `pattern.replace('*','.*')`.	3. ✓ Rewire the `build_query` call site to apply `remove_author_honorifics` to the author's `name` field.
4. ✗ Treat `mocks/mock_infobase.py` as immutable test infrastructure and refuse to consider modifying it.	4. ✓ Recognise `mocks/mock_infobase.py` as in-tree project code per `SKILL.md`'s framing of MockSite as repo-owned infrastructure.
5. ✗ Loop through query-side-only workarounds (`replace('*','')`, `[*]`, `re.escape`, no escape), each rejected on interaction with `regex_ilike`.	5. ✓ Rewrite the three `find_author` queries: escape `*` via `name.replace('*', r'\*')`; use `extract_year(...)` with year-wildcard patterns `*{year}*` for the surname branch.
6. ✗ Reopen the same alternatives a second time; never converge on a coherent plan.	6. ✓ Extend `regex_ilike` with a placeholder technique (`'\x00STAR\x00'`) so `\*` round-trips through the `*` → `.*` rewrite as a literal star.
7. ✗ Final deliberation truncated at 16K limit; loop exits due to no tool call, no `edit_file` is ever issued, the diff is empty, the patch is not submitted to the evaluator.	7. ✓ Update the wildcard-matching test to expect `None`; adjust the parametrised honorifics test to the new `str` → `str` signature.

Figure 5: Step-level trajectory comparison on SWE-bench Pro task `internetarchive__openlibrary-f343c08f`. Both agents share the same Claude Opus 4.6 backbone, the same tool set, and the same user prompt; the only difference is whether the skill document distilled during onboarding is provided. The decisive divergence is at step 4: the baseline reads `regex_ilike`, sees that it under-approximates production `ILIKE`, and treats the mock as an external constraint to route around—then enumerates query-side-only escapes that all fail and gets truncated at the 16K output limit; the skill-conditioned agent, having read `SKILL.md`'s explicit listing of `mocks/mock_infobase.py` as repo-owned project code, reaches for the symmetric two-sided fix (escape `*` on the query side, teach `regex_ilike` to round-trip `\*`) without hesitation. The downstream omission of any `edit_file` (step 7) is the same failure mode as in the qutebrowser case, but in its sharpest form: the binding constraint here is not which extension point to use but *which file is allowed to be touched at all*.

Figure 5 contrasts the two trajectories step by step: identical surface diagnosis, divergent boundary of what counts as project code.

SWE-bench Pro `5e549255` **(navidrome): the** `structs:"-"` **contract turns persistence rewrites into lookups.** The task threads a many-to-many genre relation through navidrome's persistence layer (`Album` gains a `Genres` collection, populated via the `album_genres` table and aggregated from track genres) and unifies starred retrieval behind `GetAll(filter.Starred())` in place of three per-repo `GetStarred` methods. The baseline agent produces a 132-line patch over 7 files (unresolved, 0/1 fail-to-pass): it correctly handles the model-interface, controller, and `filter.Starred()` layers, and even adds `loadAlbumGenres` to `persistence/sql_genres.go`, but stalls the moment it opens `persistence/album_repository.go`—deliberating over (i) whether `selectAlbum` or `GetAll` should carry the genre `LeftJoin + GroupBy`, (ii) how `updateGenres` should distinguish full sync from append, and (iii) which sort keys (`name`, `id`) become ambiguous once the genre table is joined—and its final long-form deliberation hits the 16K single-response limit mid-thought ("Actually, now I realize there…"). The loop exits because the truncated output contains no tool calls, without issuing a single edit on that file. As a result `albumRepository.Put` is never written, `genre_repository.go`'s legacy `album.genre = genre.name` count remains, the mocks are never extended with `GetAll`, and the only fail-to-pass test (`Rock.AlbumCount == 3`, which requires the relation table to be populated) fails. The skill-conditioned agent reaches 1/1 fail-to-pass over 25 edits in 14 files (resolved) by following three repository-specific conventions distilled into `persistence.md`: (a) the base `sqlRepository.put(id, m, colsToUpdate...)` is the canonical write path, and a typed `Put` on a concrete repository is a thin wrapper around it—this collapses the "which method writes the row" question into a one-line answer; (b) fields tagged `structs:"-"` (e.g. `Genres`, `SimilarArtists`) are excluded from `toSqlArgs` on purpose and are persisted via dedicated relation helpers, so the idiom for writing such a record is *strip the field, defer-restore it, call `r.put(m.ID, m)`, then call `r.updateGenres(...)`*—the agent writes exactly this pattern in `albumRepository.Put` without re-deriving it; (c) changing a repository interface obliges updating the matching mock in `tests/mock_*_repo.go`, since each mock is pinned by a compile-time `var _ model.XxxRepository = (*MockXxxRepo)(nil)` assertion—this triggers the otherwise easily-forgotten `GetAll` additions on the artist and mediafile mocks. The case crystallises a complementary face of the central thesis: when the bottleneck is not which extension point to use but how to *compose* an existing one with three implicit invariants, the skill turns a multi-step reverse-engineering task into a sequence of confident lookups.

Figure 6 contrasts the two trajectories step by step: identical plan, divergent persistence-layer execution.

SWE-bench Pro `233cb1cc` **(qutebrowser): when an over-confident skill skips the exploration that would have caught it.** The task asks for a new `overlay` value for the `scrolling.bar` setting that emits the Chromium flag `-enable-features=OverlayScrollbar` only when the backend is QtWebEngine, Qt $\geq$ 5.11, and the platform is not macOS, with the legacy boolean migration `False` $\rightarrow$ `overlay`, and an `@js_headers` pytest marker for the end-to-end header tests. This is one of the rare cases (a $\sim$4% regression rate; 8 of 200 task-runs in our 4-run pool, see Figure 3) where the skill-conditioned agent regresses relative to the baseline: the baseline produces a resolved patch over 8 files, while the skill-conditioned agent produces an unresolved patch over 6 files whose core logic is functionally close but whose peripheral plumbing is wrong in three correlated ways. The mechanism is a *tunnel effect*: because the skill-conditioned agent has loaded `SKILL.md`, `config_changes.md`, and `testing_patterns.md` before opening any source file, it commits early to the patterns those documents codify, and skips the exploratory `grep` pass that the baseline performs over `is_mac`, the two `conftest.py` files, and `pytest.ini`. Three repository-specific facts that exploration would have surfaced are silently elided by the skill's priors: (a) the canonical macOS guard in this codebase is `utils.is_mac`, not `sys.platform.startswith('darwin')`—but the skill's worked example for `configinit.py` happens to use `sys`, which is already imported, so the skill-conditioned agent never adds the `utils` import and never discovers that the established idiom would also yield a more reliable monkeypatch target; (b) the `js_headers` marker handler belongs in `tests/end2end/conftest.py`, where the existing markers-list dispatch loop lives, not in the global `tests/conftest.py`—the skill mentions end-to-end tests only generically and provides the `js_prompt` pattern as a reference, which lives in the global conftest and which the agent therefore mirrors verbatim; (c) every new pytest marker must be registered in `pytest.ini` to avoid an unknown-marker warning, but the skill never names `pytest.ini` and the agent, satisfied that the marker handler exists, never thinks to look. The `testing_patterns.md` file addition-

Baseline agent 0/1 f2p, unresolved ✗	**Skill-conditioned agent** 1/1 f2p, resolved ✓
1. ✓ Read model interfaces, `filter/filters.go`, `genre_repository.go`, `sql_genres.go`; correctly plan the four-quadrant change set.	1. ✓ Read `SKILL.md` + `persistence.md` + `subsonic_api.md` before opening any source file.
2. ✓ Drop `GetStarred` from `model/album.go`, `model/artist.go`, `model/mediafile.go`.	2. ✓ Drop `GetStarred` from all three model interfaces; add `Genres` field and `Put(*Album) error` to `AlbumRepository`.
3. ✓ Add `Starred()` in `server/subsonic/filter/filters.go`; switch the `album_lists.go` controller to `GetAll(filter.Starred())`.	3. ✓ Add `filter.Starred()` and switch `album_lists.go` + `getAlbumList` to `GetAll(filter.Starred())`.
4. ✓ Add `loadAlbumGenres` to `persistence/sql_genres.go`, mirroring `loadMediaFileGenres`.	4. ✓ Implement `albumRepository.Put` with the strip-then-defer-restore pattern: `m.Genres = nil; defer ...; r.put(m.ID, m); r.updateGenres(...)` per `persistence.md`'s `structs:"-"` rule.
5. ✗ Open `album_repository.go`; deliberate over whether `selectAlbum` or `GetAll` carries the genre `LeftJoin + GroupBy`, and how `squirrel.From()` interacts with `CountAll`.	5. ✓ Tighten `selectAlbum` to `Columns("album.*")`; extend `GetAll` with the genre `LeftJoin + GroupBy("album.id")` and a `loadAlbumGenres` post-pass; route `FindByArtist` and `GetRandom` through `GetAll`.
6. ✗ Spot the deletion bug in `updateGenres` but cannot decide between a custom `updateAlbumGenres` and reusing the shared helper.	6. ✓ Fix `updateGenres` to delete all existing links (full sync); rewrite `refresh` to aggregate track genres via `media_file_genres` and dispatch through the new `Put`.
7. ✗ Notice `name` / `id` ambiguity in `AlbumsByYear` / `AlbumsByGenre` sorts once the genre table is joined; loop through qualified-name fixes without committing.	7. ✓ Replace `genre_repository.GetAll`'s legacy `album.genre = genre.name` subquery with `LeftJoin album_genres a + count(distinct a.album_id)`.
8. ✗ Final thought truncated at 16K limit; loop exits due to no tool call, `album_repository.go` is never edited, mocks are never updated, `genre_repository.go`'s legacy `album.genre = genre.name` count remains, f2p `Rock.AlbumCount == 3` fails.	8. ✓ Add `GetAll` to `tests/mock_artist_repo.go` and `tests/mock_mediafile_repo.go` per `persistence.md`'s mock-sync rule; update the three persistence test files.

Figure 6: Step-level trajectory comparison on SWE-bench Pro task `navidrome__navidrome-5e549255`. Both agents share the same Claude Opus 4.6 backbone, the same tool set, and the same user prompt; the only difference is whether the skill document distilled during onboarding is provided. The two trajectories agree perfectly on the model-interface, controller, and `filter.Starred()` layers (steps 1–3), and the divergence is concentrated in the persistence layer: at step 4 the baseline opens `album_repository.go` and re-derives `put`, `toSqlArgs`, and `updateGenres` from scratch, losing its final turn to a 16K output truncation while deliberating over which question to answer first; the skill-conditioned agent has those answers preloaded—`Put` is a wrapper around base `put`, `Genres` is a `structs:"-"` field that needs strip-then-restore, mocks must track interface changes—and proceeds linearly through 14 files. The case is, in this sense, the structural mirror of the `qutebrowser` case: there the skill supplied the *location* of the load-bearing extension point; here it supplies the three-step *recipe* for using one whose pieces are scattered across the persistence layer.

ally over-specifies the `version_check` lambda signature as `lambda version, exact=False, compiled=True: new_qt`, which the skill-conditioned agent dutifully reproduces in places where the simpler `lambda version, compiled=False: True` used by the existing `reduce_args` fixture would have been sufficient. The case crystallises the symmetric counterpart of the §4 thesis:

skills compress repository-specific lookups into confident priors, and that compression is overwhelmingly net-positive, but it occasionally substitutes confidence for the exploration that would have caught a generic prior firing in the wrong place—matching the body-text observation that a small minority of task-runs regress under skill conditioning even when the baseline succeeds.

Baseline agent resolved ✓	**Skill-conditioned agent** unresolved ✗
1. ✓ Open `configdata.yml`, `configfiles.py`, `configinit.py` cold; trace the existing `scrolling.bar` call sites and the `_qtwebengine_args()` flag-yield loop.	1. ✓ Read `SKILL.md` + `config_changes.md` + `testing_patterns.md` before opening any source file.
2. ✓ Add `overlay` to `valid_values`; flip the boolean migration to `False` → `overlay`.	2. ✓ Add `overlay` to `valid_values`; flip the boolean migration to `False` → `overlay`.
3. ✓ Grep `is_mac` across the tree; discover the project-wide idiom `utils.is_mac` and add `from qutebrowser.utils import utils` to `configinit.py`.	3. ✓ Follow `config_changes.md`'s "add to the settings dict, NOT as separate code" rule; place the `overlay` entry inside the `settings` dict in `_qtwebengine_args()` (closer to oracle than the baseline's post-loop block).
4. ✓ Append a guarded `if / yield` block after the settings loop: `if scrolling.bar == 'overlay' and qtutils.version_check('5.11', compiled=False) and not utils.is_mac: yield '-enable-features=OverlayScrollbar'`.	4. ✗ Reuse the already-imported `sys` for the macOS guard (`not sys.platform.startswith('darwin')`) because the skill's `configinit.py` example never mentions `utils.is_mac`; never grep for `is_mac`.
5. ✓ Inspect both `tests/conftest.py` and `tests/end2end/conftest.py`; recognise the markers-list dispatch loop in the e2e conftest as the right insertion point for `js_headers`.	5. ✗ Add the `js_headers` marker handler to the global `tests/conftest.py`, mirroring the skill's `js_prompt` reference, instead of the e2e conftest's markers-list dispatch loop.
6. ✓ Register `js_headers` in `pytest.ini` alongside the other markers, avoiding the unknown-marker warning.	6. ✗ Skip `pytest.ini` entirely—the skill never names it, and the agent stops exploring once the handler exists.
7. ✓ Add a parametrised `test_overlay_scrollbar` using `monkeypatch.setattr(configinit.utils, 'is_mac', ...)` and the simple `lambda version, compiled=False: ...` signature already used by `reduce_args`.	7. ✗ Reproduce `testing_patterns.md`'s mandated lambda signature `lambda version, exact=False, compiled=True: ...` verbatim; monkeypatch `configinit.sys.platform` (a module-level reference, fragile) instead of `configinit.utils.is_mac`.
8. ✓ Submit; all gating, migration, and marker tests pass.	8. ✗ Submit; `js_headers` marker is unrecognised and routed through the wrong conftest; e2e marker test and platform-mock test fail.

Figure 7: Step-level trajectory comparison on SWE-bench Pro task `qutebrowser__qutebrowser-233cb1cc`. Both agents share the same Claude Opus 4.6 backbone, the same tool set, and the same user prompt; the only difference is whether the skill document distilled during onboarding is provided. The decisive divergence is at step 3–6: the baseline, having no prior, performs three exploratory greps—`is_mac` across the tree, both `conftest.py` files side by side, and `pytest.ini`—and from each grep recovers a repository idiom; the skill-conditioned agent has a confident-but-incomplete prior for each of the three (`sys.platform` from the worked example, the global conftest from the `js_prompt` reference, no mention of `pytest.ini` at all) and so never runs those greps. The case is the structural mirror of the `f8e7fea0` case: there the skill supplied the load-bearing extension point that pre-training priors miss; here the skill substitutes a too-confident prior for the exploration that would have caught a generic prior firing in the wrong place.

Figure 7 contrasts the two trajectories step by step: identical core logic, divergent peripheral plumbing because the skill replaced exploration with a too-confident prior.

Lesson on memory accumulation: chronological order matters. Surfacing the case above also exposed a non-obvious failure mode of naive memory accumulation. In an early version of our pipeline, learning commits were ordered by LLM-judged relevance to the upcoming task rather than by commit timestamp. Under that ordering the agent would first study a recent commit that already used the `Callable` indirection idiom, distil it into the skill, and then encounter an older commit that pre-dated the convention and lacked it—which the contrastive-reflection step misread as an instance of a *hallucinated* convention and removed from the skill, undoing exactly the knowledge that mattered for the held-out task. After switching to strict chronological order (see §3.2), each contradiction is interpreted as the project *adopting* a new convention rather than the agent fabricating one, and the `Callable` pattern is preserved through the full learning sweep. This is one concrete reason our framework treats the temporal ordering of commits as a first-class part of the memory mechanism, not merely a presentation detail.

I Implementation Details

The sandbox image we hand to the agent ships a generic, project-agnostic MCP toolset—web browser automation, image search and generation, speech and sound synthesis, structured data sources, and website deployment, on top of the usual code- and file-manipulation tools. The LearningToCommit pipeline does not gate this set per phase; it only ever invokes the seven tools listed in Tab. 13, and we limit the schema documentation to those. What does differ across phases is the set of directories the workflow mounts into each phase's sandbox, which is what bounds where the file-I/O tools can read and write; Tab. 14 records this layout.

I.1 Active Tool Schema

The seven tools actually invoked by the agent are listed in Tab. 13. Five (`read_file`, `edit_file`, `write_file`, `shell`, `ipython`) handle code and file manipulation; the remaining two (`todo_read`, `todo_write`) maintain the agent's own multi-step plan.

Table 13: Active tools invoked by the agent across the six phases of the LearningToCommit pipeline. Required parameters are typed: `s`=string, `a`=array. Optional parameters are omitted for brevity.

Tool	Required parameters	Description
`todo_read`	—	Read the agent's current TODO list.
`todo_write`	`todos:a`	Create or update the structured TODO list used for multi-step planning.
`read_file`	`file_path:s`	Read a file by absolute path; binary office formats (PDF, DOCX) are auto-converted to Markdown.
`edit_file`	`file_path:s`, `old_string:s`, `new_string:s`	Exact-string replacement in an existing file (must `read_file` first).
`write_file`	`file_path:s`, `content:s`	Overwrite or append to a file (content capped at 100K characters per call).
`shell`	`command:s`	Execute one shell command in a non-persistent session (no state carries between calls).
`ipython`	`code:s`	Execute Python in a stateful Jupyter-style REPL; variables and imports persist across calls; the `!` prefix runs bash.

I.2 Phase Workflow and Sandbox Mounts

Tab. 14 summarises, for each of the six phases, the tool calls that dominate the agent's trajectories, the sandbox directories that bound those calls, and the artifact the workflow expects after the phase finishes.

A few signatures in the trajectory data merit note. (i) The directory each phase writes to is itself a phase identifier: `workspace/` for solve and learn-attempt, `skills/` for learn-reflect and

Table 14: Per-phase agent tool usage. "Primary actions" lists the tool calls that dominate each phase's trajectories; "Mount" indicates which sandbox directories the phase can read/write (`(R)` marks read-only mounts; otherwise read–write); "Key artifact" is the file the workflow expects after the phase finishes.

Phase	Primary actions	Mount	Key artifact
`learn_attempt`	`read_file`, `edit_file`, `write_file`, `shell`	`workspace/`, `skills/` (R)	`output/changes.diff`
`learn_reflect`	`read_file`, `write_file`, `edit_file`	`skills/`	updated `skills/`
`merge_skills`	`read_file`, `write_file`, `edit_file`	`skills/`	merged `skills/`
`solve_with_skill`	`read_file`, `edit_file`, `write_file`, `shell`	`workspace/`, `skills/` (R)	`output/changes.diff`
`solve_baseline`	`read_file`, `edit_file`, `write_file`, `shell`	`workspace/`	`output/changes.diff`
`judge`	`read_file`, `write_file` (`result.json`)	`workspace/` (R), `judge/`	`judge/result.json`

merge, `judge/result.json` for judge. (ii) `learn_attempt` and `solve_with_skill` share an identical inner workflow—"read `SKILL` $\rightarrow$ explore repo $\rightarrow$ edit code"—and differ only in that `learn_attempt` is followed by a separate reflection step that writes lessons back to the skill tree. (iii) `merge_skills` has the narrowest tool footprint (file I/O only): no shell, no IPython, no workspace access, since merging skill notes is a pure skill-space operation. (iv) `judge` is the only phase that never modifies code: it treats `workspace/` purely as a reference for cross-checking `api_reuse` and `style_consistency` between the agent's diff and the oracle diff.

I.3 Context Compaction

Long agent trajectories accumulate large file reads, search dumps, and shell stdout in successive tool responses, which can fill the model's context window before a phase finishes. We adopt a lightweight rule-based compactor that preserves the agent's reasoning skeleton while aggressively pruning stale tool outputs. The compactor is invoked unconditionally after every LLM round in the agent loop; whether it actually rewrites messages depends on the token-budget threshold below.

Trigger. Let $L = 262{,}144$ be the context-window budget (256K tokens), $\tau = 0.8$ the trigger threshold, and u the current message-list token usage (computed jointly over the message list and the tool schemas). The compactor begins rewriting messages when $u/L \geq \tau$, or when adding one more 16K-token response would overflow the budget; otherwise it returns the messages unchanged.

Two-stage rewrite. Compaction operates on *tool-call rounds*—the unit consisting of one assistant message bearing `tool_calls` together with all of its corresponding tool-response messages—rather than on individual messages, so that `tool_call_id` pairings remain syntactically valid throughout. **Stage 1** replaces the `content` field of every tool-response message in *all but the last 3 rounds* with a fixed placeholder string ("previous tool_call hidden") and tags the round with `compact: true` to prevent re-processing on subsequent rounds. **Stage 2** runs only if usage after Stage 1 still exceeds the threshold; it extends the redaction inward, leaving only the most recent round's tool responses intact.

The compactor never deletes messages, never modifies user/system/assistant content, and never invokes an LLM to summarise. The trade-off is intentional: rule-based rewriting is deterministic and essentially free, but redacted tool outputs are lost rather than summarised, so the agent cannot subsequently re-read them. Empirically this is acceptable because tool responses (file dumps, search

results, shell stdout) carry quickly-decaying value, whereas the assistant's interleaved reasoning—which encodes what was learned from those outputs—is preserved verbatim.

I.4 Pairwise Win Rate Computation

For each task, we run a four-round ABBA schedule in which each agent is presented as A in two rounds and as B in the other two, controlling for the judge's position bias. The LLM judge returns one of three verdicts—win, loss, or tie—per dimension per round. We collapse the four dimensions and the four ABBA rounds into a single per-task score by averaging, and aggregate to the configuration-level number reported in Table 2 via the standard "win plus half-tie" formula

$$\text{WinRate} \;=\; \frac{\#\text{wins} + 0.5 \cdot \#\text{ties}}{\#\text{wins} + \#\text{losses} + \#\text{ties}},$$

computed against the matched no-skill / Vanilla reference. The judge itself follows the same two-stage decoding pattern used in our data-curation pipeline (cf. §3.4): free-form analysis with Gemini 3.1 Pro followed by structured-JSON extraction with Claude Sonnet 4.5. Under this convention, 50% corresponds to indistinguishable from the reference; a strict $> 50\%$ value indicates a net preference for the skill-conditioned (or non-reference) configuration.

I.5 Compute-Matched Baselines and Token Accounting

To address whether the observed gains merely stem from spending more compute, we introduce a forced-exploration baseline where the vanilla agent is instructed to explore thoroughly and prevented from submitting prematurely. On the main benchmark, it runs until roughly matching the skill agent average steps. On SWE-bench Pro, it runs until roughly reaching the exploration depth the skill agent exhibits on helped tasks against the scenario in Figure 3 panel b. Table 15 presents the results. A blank compute budget buys no quality across both testbeds. Forcing the agent to spend 1.5 times the steps on the main benchmark drops localisation by 6.2 percentage points and stalls the judge at a statistical tie, and its only improvement in line deviation does not win judge approval. On SWE-bench Pro, pushing vanilla to the helped-task exploration depth leaves pass@1 within the inter-run standard deviation of Table 3, while the skill agent adds only 2.0 steps per task for a +8.0 percentage point gain. Knowing where to explore is the key, rather than merely exploring deep.

Table 15: Forced-exploration baseline (Claude Opus 4.6). The vanilla agent is instructed to explore thoroughly and is prevented from early submission: on the Learning to Commit benchmark until it approximately matches the skill agent's average steps, and on SWE-bench Pro until it approximately reaches the exploration depth the skill agent shows on helped tasks (Figure 3b). Judge net-win is win rate minus loss rate against Vanilla over 200 pairwise rounds.

	Learning to Commit benchmark				**SWE-bench Pro**	
Condition	**File IoU**	**Steps**	**Line Dev**	**Judge net-win**	**pass@1**	**Steps**
Vanilla	62.0%	26.0	0.58	—	48.0%	30.6
Vanilla + forced exploration (no skill)	55.8%	39.1	0.47	+2.5 pp (tie: 85W/80L/35T)	46.0%	43.5
Ours (skill-conditioned)	**67.0%**	40.0	**0.43**	+18.0 **pp**	**56.0%**	32.6

Table 16 records the average completion tokens per task during the solve phase. Average steps grow from 26 to 40 on the main benchmark, but completion tokens only increase by 6%, because the skill agent primarily changes trajectories on the minority of tasks where it flips the outcome, and the extra steps are mostly short targeted reads. Tool steps actually overestimate our compute footprint. Our agent takes the most steps in Table 2 but consumes fewer completion tokens than unsupervised self-reflection, because skill-guided steps are quick read-edit actions, whereas free-form reflection writes long essays at every step. On SWE-bench Pro, the skill agent adds 2.0 tool steps per task (an increase of about 6.5%), and roughly 1.5 of those steps are spent reading the skill document itself. Repository onboarding represents a one-time cost of 0.6 to 0.9 million completion tokens per repository.

I.6 Factual Audit of the Learned Skill Documents

We slice the learned skill documents into the finest-granularity items by their markdown structure, filtering out pure navigation text and fragments, and perform stratified sampling by repository to

Table 16: Average solve-phase completion tokens per task (Claude Opus 4.6).

Method (solve phase)	Completion tokens per task
Learning to Commit benchmark	
Vanilla	26.3K
Passive RAG	27.4K
Unsupervised self-reflection	30.5K
Ours	28.0K
SWE-bench Pro	
Vanilla → Ours	15.8K → 17.0K

extract 60 items from the pool. We select Claude Fable 5 as the audit model because it is stronger than the generating Claude Opus 4.6 model to maximise audit accuracy, and using a different model reduces self-preference bias. The extended thinking configuration is set to the highest level. For each item, we programmatically collect evidence from the corresponding repository checkout, verifying file paths via `git ls-files` and extracting identifiers along with context snippets via `git grep`. We provide the item and the evidence to the audit model and require a three-way verdict among factually correct, incorrect, and cannot verify. Every verdict must cite specific evidence including file paths and symbols from the repository, and baseless verdicts are not allowed. The language model serves only as a first pass, and the authors manually verify all flagged items.

The audit model directly verifies 75% of the items as factually correct. Manual review of the remaining flagged items confirms a hard factual error rate of 5% (3 out of 60 items), while the rest are audit false positives or harmless API renames. A subtler over-specialisation failure mode is discussed in Appendix A. The full auditor prompt is reproduced below.

```
You are auditing one entry from a machine-written engineering skill
document for the `{repo}` repository. The document was distilled from
real development experience on this repository. The evidence below was
mechanically extracted (git ls-files / git grep with surrounding
context) from a checkout of the repository.

## Skill entry (from {source_file}, section "{heading}")
<entry>
{text}
</entry>

## Evidence from repository checkout
{evidence}

## Your task
Judge the factual accuracy of the entry's checkable claims (file paths,
API/function/class names, described behaviours, orderings, conventions)
against the evidence.

Verdicts:
- "factually_correct": repo-specific claims are consistent with the
  evidence, or the entry is generic engineering advice that the
  evidence does not contradict.
- "incorrect": a repo-specific claim is contradicted by the evidence in
  a way that would actively mislead a developer (wrong mechanism, wrong
  behaviour, wrong relationship).
- "cannot_verify": the evidence is insufficient to confirm or refute
  the claim.

Rules:
- The checkout may postdate the learning period; renames, moves, and
  refactors alone are not counted as factual errors.
- Cite concrete evidence (file paths and symbols) for every verdict.
  No ungrounded verdicts are allowed.
- Output JSON: {"verdict": ..., "confidence": "high"|"low",
  "reason": ..., "evidence_cited": [...]}.
```

I.7 Compute and API Cost

All experiments run the agent against hosted LLM APIs. We use no local GPU compute. We report API spend for Claude Sonnet 4.6 and Claude Opus 4.6, both accessed with thinking effort set to `high` (the configuration used for every with-skill / baseline run reported in §4). Aggregate API spend over the entire research project—reported runs together with preliminary, failed, and ablation experiments, plus the additional baselines and human studies added during the review period—exceeds USD 20,000. Because compute is API-only, reporting reduces to dollar spend rather than GPU-hours.

J System Prompts

This appendix lists the verbatim text of all system prompts used by our pipeline. Each prompt is loaded directly from the project repository at compile time (no hard-coded duplicate is maintained in the paper source). Curly-brace placeholders (e.g., `{subject}`, `{patch}`, `{categories_json}`) are filled at runtime by the orchestration code.

J.1 Commit suitability filter (`wash_suitable.txt`)

```
You are an evaluator of the learning value of code-repository commits. We are conducting
research on online learning. The background is that we believe traditional repo development
contains a lot of development details and conventions that cannot be derived from static
file states alone - they need to be learned from the history of commit operations. For
example, when facing a bug, how to fix it can be learned from the patch of the real
historical commit corresponding to that bug. Therefore, we now need to judge, for the
commit below, whether it has the potential to teach some continuously-accumulable, valuable
experience, so that future development can better match the habits of a senior developer of
the repo.

Below is the patch of a git commit. Please judge whether this commit is suitable as
learning material for "repository development pattern learning".

Good learning material is typically: substantial code changes that exhibit the repo's
architectural design / development conventions / feature-implementation patterns.
Unsuitable material is typically: pure config changes, documentation updates, dependency
upgrades, auto-generated code, or formatting/renaming.
```

J.2 Cluster discovery (`wash_cluster.txt`)

```
We are conducting research on online learning. The background is that we believe
traditional repo development contains a lot of development details and conventions that
cannot be derived from static file states alone - they need to be learned from the history
of commit operations. For example, when facing a bug, how to fix it can be learned from the
patch of the real historical commit corresponding to that bug.

From a code repository, we have already used an LLM to filter out a batch of high-quality
commits. Each one includes an estimate of what continuously-accumulable experience can be
learned from that commit.

Now I need to categorize the learnable experience estimated for these commits, summarizing
them into several different development categories - grouping the experience that can be
continuously learned, such as architectural-design learning, certain feature-implementation
patterns, debugging/fixing experience, etc.

Please look through all of the commit-experience-estimate descriptions below, then
summarize them into 4-8 categories. The categories should be as non-overlapping as
possible, and should serve as different aspects of experience to accumulate via online
learning. Each category should contain:
- name: a short English label (snake_case)
- description: an English description explaining what kind of experience this category
mainly covers
```

J.3 Commit classification (`classify.txt`)

```
We are conducting research on online learning. The background is that we believe
traditional repo development contains a lot of development details and conventions that
cannot be derived from static file states alone - they need to be learned from the history
of commit operations. For example, when facing a bug, how to fix it can be learned from the
patch of the real historical commit corresponding to that bug.

We have already used an LLM to filter a batch of commits with learning value from the
repository, and estimated for each commit what experience can be learned. Then we
summarized these experiences into several development categories. Now we need you to assign
each commit to the single best-matching category.

Below is a git commit:

commit subject: {subject}
S1-stage estimate of this commit's learning value: {reason}

Below is the patch of this commit:
========== patch begin ==========
{patch}
========== patch end ==========

Please choose the single best-matching category from the list below (choose only one):

{categories_json}

Combine the commit subject, the learning-value estimate, and the patch content to choose
the most appropriate category.
```

J.4 Synthetic query generation (`synthetic_query.txt`)

```
We are conducting research on online learning. The background is that we believe
traditional repo development contains a lot of development details and conventions that
cannot be derived from static file states alone - they need to be learned from the history
of commit operations. For example, when facing a bug, how to fix it can be learned from the
patch of the real historical commit corresponding to that bug.

Now I need you to use the information of the git commit below to reverse-synthesize a "task
description" (in the style of an issue / task ticket). This task description will be given
as input to an AI Agent so that it can implement the corresponding change in the code
repository.

Requirements:
- Write in English, in the style of an issue description written by a project manager or
development lead
- Describe "what needs to be done" and "why it needs to be done", but do not reveal the
concrete implementation (do not mention specific function names, class names, or file
paths)
- A developer familiar with the repo should, after reading it, understand the requirement
and be able to implement it

commit subject: {subject}
commit body: {body}

Below is the patch of this commit:
========== patch begin ==========
{patch}
========== patch end ==========

Now, based on the git commit information above, reverse-synthesize a "task description" (in
the style of an issue / task ticket). This task description will be given as input to an AI
Agent so that it can implement the corresponding change in the code repository. Please
output the task description text directly.
```

J.5 Top-k historical commit selection (`select_learning.txt`)

```
We are conducting research on online learning. The background is to let an AI Agent first
learn development experience from the repository's historical commits, and then go on to
solve a new issue. Now we need your help to select the most worthwhile historical commits
to learn from.

Below is a GitHub issue to be resolved:

========== issue begin ==========
{problem_statement}
========== issue end ==========

Below is a batch of historical commits with learning value from this repository (already
filtered for quality). Each commit includes a subject, description, and a summary of the
code changes:

========== candidate commit list ==========
{candidates}
========== candidate list end ==========

Please select the {top_k} commits that have the most learning value for solving the issue
above.

Selection criteria:
- The change touches similar modules, files, or subsystems
- It exemplifies similar bug-fix patterns or feature-implementation styles
- It involves related internal APIs, helper functions, or architectural patterns
- It can help the Agent understand the development conventions of this part of the code

Please give a detailed analysis of your reasoning for the selection.
```

J.6 Learning-phase blind attempt (`learn_attempt.txt`)

```
You are a senior code developer. Your task is to read the task description below,
understand the current state of the repo, and then make targeted code changes to implement
the required functionality.

- **Workspace** at `/mnt/agents/workspace/` - the repository source code; you should make
your changes on top of the code in this directory.
- **Skills directory** at `/mnt/agents/skills/` - development experience you have
accumulated from this repository's history; for your reference only. Do not rely on past
experience entirely, since experience can become outdated. Focus on the concrete needs of
the current task, and use past experience to help you understand the development of the
current task.
- After you finish modifying the code, an automated workflow will export your changes with
the following commands:
```bash
BASE=$(git rev-list --max-parents=0 HEAD)
git add -N --all .
git diff $BASE --no-color --binary > /mnt/agents/output/changes.diff
```

If `.gitignore` hides files you create or modify, you can use `git add -f <file>` to
force-track them.

**Important: only run `git add -f` on files you actually created or modified. Do not
force-add existing files you have not modified - they are gitignored for a reason. Adding
unmodified files bloats the patch and lowers the score.**

Below is your current task description

{task description}
```

J.7 Learning-phase contrastive reflection (`learn_reflect.txt`)

```
You just attempted to implement a task. Now compare your implementation with the real
commit for that task (the oracle diff).

## Real implementation (Oracle Diff)

commit: {sha_short} - {subject}

```diff
{real_diff}
```

Please compare your earlier implementation with the oracle diff above, and from the
differences extract **experience that is specific to this repository and useful for future
development**, then update it under the `/mnt/agents/skills/` directory.

Our goal is to continuously accumulate development experience for this repository - for
example, internal API usage, file organization conventions, architectural patterns, and
common ways of making changes. This experience will help you complete similar tasks faster
and more accurately in the future. For overly minor differences (such as variable-naming
preferences, comment style, etc.), you can ignore them.

Update the experience under `/mnt/agents/skills/`; `SKILL.md` is the core file. If the
content is too long, you may split it into sub-files at your discretion, putting different
aspects of experience into different files or directories, keeping a clean hierarchy. Read
the existing skill files thoroughly before changing them.

Additionally, while analyzing, if you find that the differences between your current
implementation and the oracle diff actually arose because you were misled by some past
skill, then consider deleting those previously accumulated incorrect skills, so they do not
trip you up next time.
```

J.8 Skill merge for parallel learning (`merge_skills.txt`)

```
We are conducting research on online learning. The background is that we believe
traditional repo development contains a lot of development details and conventions that
cannot be derived from static file states alone - they need to be learned from the history
of commits. We let you learn the repository's historical commits one by one, extracting
from each commit experience that may be useful for future development, and continuously
accumulating it into a skills directory.

You have just learned from two separate batches of commits and produced two independent
skills directories. Now you need to summarize and merge them into a single one.

- `/mnt/agents/skill_a/` - from earlier commits
- `/mnt/agents/skill_b/` - from later commits

Please first read all the files in both directories thoroughly, then merge them into
`/mnt/agents/skills/`. When merging, note:
- Deduplicate and consolidate - when both sides describe the same thing, merge into a
single entry, keeping the more precise version.
- When there is a conflict, prefer the version from skill_b (from a newer commit, more
likely to reflect the current state of the repo).
- Do not drop a piece of experience just because it appears on only one side; different
commits touch different subsystems, and each has its own lessons.
- You may reorganize the directory structure as you see fit. `SKILL.md` is the core file;
if the content becomes too long, you may split it into sub-files at your discretion,
putting different aspects of experience into different files or directories, keeping a
clean hierarchy.
```

J.9 Solve phase: baseline without skills (`solve_baseline.txt`)

````
You are a senior code developer. Your task is to read the task description below,
understand the current state of the repo, and then make targeted code changes to implement
the required functionality.

- **Workspace** at `/mnt/agents/workspace/` - the repository source code; you should make
your changes on top of the code in this directory.
- After you finish modifying the code, an automated workflow will export your changes with
the following commands:
```bash
BASE=$(git rev-list --max-parents=0 HEAD)
git add -N --all .
git diff $BASE --no-color --binary > /mnt/agents/output/changes.diff
```

If `.gitignore` hides files you create or modify, you can use `git add -f <file>` to
force-track them.

**Important: only run `git add -f` on files you actually created or modified. Do not
force-add existing files you have not modified - they are gitignored for a reason. Adding
unmodified files bloats the patch and lowers the score.**

Below is your current task description

{task description}
````

J.10 Solve phase: skill-conditioned (`solve_with_skill.txt`)

````
You are a senior code developer. Your task is to read the task description below,
understand the current state of the repo, and then make targeted code changes to implement
the required functionality.

- **Workspace** at `/mnt/agents/workspace/` - the repository source code; you should make
your changes on top of the code in this directory.
- **Skills directory** at `/mnt/agents/skills/` - development experience you have
accumulated from this repository's history; for your reference only. Do not rely on past
experience entirely, since experience can become outdated. Focus on the concrete needs of
the current task, and use past experience to help you understand the development of the
current task.
- After you finish modifying the code, an automated workflow will export your changes with
the following commands:
```bash
BASE=$(git rev-list --max-parents=0 HEAD)
git add -N --all .
git diff $BASE --no-color --binary > /mnt/agents/output/changes.diff
```

If `.gitignore` hides files you create or modify, you can use `git add -f <file>` to
force-track them.

**Important: only run `git add -f` on files you actually created or modified. Do not
force-add existing files you have not modified - they are gitignored for a reason. Adding
unmodified files bloats the patch and lowers the score.**

Below is your current task description

{task description}
````

J.11 Pairwise A/B judge (judge_pairwise.txt)

```
You are a senior code review expert. There is a PR task; the real developer has provided a
reference answer (the Oracle diff). Two AI developers (A and B) have each produced their
own code changes.

Your task is to compare the patch quality of A and B along 4 specific dimensions.

## Inputs

- **Task description**: the requirement received by the developer
- **Oracle Diff**: the real commit, used as a reference standard for the scope and intent
of the change
- **Developer A's Diff**
- **Developer B's Diff**

Note: the Oracle diff is only a reference standard, not the only correct implementation. A
and B may use different approaches, but you need to judge who is overall closer to the
"correct and native" implementation that the repo expects.

## Evaluation dimensions

### Q1. Scope Alignment

Whose modified files, classes, and function locations more precisely correspond to the
Oracle's modification scope? Who avoided going off-track and modifying unrelated code?

### Q2. Logic Similarity

Setting aside coding style, regarding the core logic that the Oracle uses to solve the
problem, whose implementation is closer in approach? Different implementation styles are
allowed, but the core angle for solving the problem must match.

### Q3. Redundancy and Hallucination

Whose implementation has fewer of the following issues:

- Over-engineering (adding extra functionality the Oracle did not require)
- Large-scale refactoring (modifying lots of code the Oracle did not touch)
- Hallucinated code (introducing APIs / fields / paths / call patterns that do not exist in
the repo)
- Unnecessary defensive branches, boilerplate, or duplicated logic

### Q4. Code Style

Whose variable naming, comment conventions, and type-hint style better matches the
repo-native style shown in the Oracle?

## Analysis requirements

Please analyze **each** of the four dimensions Q1-Q4 in turn. For every dimension you must:

1. Write 2-4 sentences of concrete analysis, clearly pointing out the strengths and
weaknesses of A and B
2. Explicitly give the winner of that dimension: `A_WIN` / `B_WIN` / `TIE`

After analyzing all four dimensions, give:

- A 2-4 sentence overall summary
- The final winner: `A_WIN` / `B_WIN` / `TIE`
```

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: Claims and scope are stated in §1 and §4.1; experimental support is in §4.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Limitations are discussed in Appendix A.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

Justification: The paper does not include theoretical results.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The agent loop, tool schema, model configuration, and full prompts are documented in Appendices I and J.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: The supplementary material includes the Learning to Commit benchmark data, the SWE-bench Pro 50-task subset, the evaluation and judging pipeline (prompts, scoring formula, sandbox harness), and the agent prompts used in our experiments. The agent framework we used for the experiments runs on internal infrastructure and cannot be released. We have instead released an open-source version built on the public Claude Agent SDK at `https://github.com/LearningToCommit/LearningToCommit`. It follows the same workflow and comes with instructions for running the learning and evaluation steps. Because it is a reimplementation rather than the exact code behind the reported numbers, rerun results may differ slightly.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification: Experimental setup, baselines, and metrics are described in §4.1 and §3.5, with full implementation details in Appendix I.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: Table 3 reports mean ± 1 SD across 4 independent agent rollouts on SWE-bench Pro (paired per-run differences for Δ). Figure 3(a) shows the 4-run pooled outcome counts with per-run scaled SD error bars on the helped/hurt bars; Panel (b) reports

± 1 SEM across pooled per-task trajectories within each outcome group. The full organicity benchmark in Table 2 reports single-run performance without error bars due to compute constraints (see Appendix I.7); its agent-model ablation block already spans three independent backbones (Kimi K2.5, Claude Sonnet 4.6, Claude Opus 4.6), which provides built-in cross-model robustness against run-to-run noise.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: API spend and compute setup are reported in Appendix I.7.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research follows the NeurIPS Code of Ethics. Both benchmarks are built from public open-source repositories; the internal repository used in Appendix D is not released. The human studies in Appendix D only ask experienced engineers to review code patches, collect no personal information, and release only anonymised verdicts. We release no high-risk models or data.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.

- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Positive and negative societal impacts are discussed in Appendix B.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper does not release pre-trained models, image generators, or scraped datasets with high misuse risk; the released benchmarks are curated from publicly available open-source repositories.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: All used assets (SWE-bench Pro, the five source open-source repositories, and the LLM APIs) are cited; their licenses and terms of use are respected.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: The Learning to Commit benchmark, the SWE-bench Pro 50-task subset, the evaluation scripts, and the human study materials are released as supplementary material with documentation. The open-source agent framework, built on the public Claude Agent SDK, is released with documentation at `https://github.com/LearningToCommit/LearningToCommit`.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [Yes]

Justification: No crowdsourcing platform was used. Appendix D describes who took part and how the reviews were run: the LLM-judge check uses in-house expert annotators, and the mergeability study uses two maintainers of an internal repository (Track A) and three outside engineers invited through the authors' professional networks (Track B). The supplementary material includes the full instructions given to the Track B annotators and the review protocol for Track A. Participants worked on code files and spreadsheets, so there are no screenshots. All participants took part as volunteers and received no payment.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [No]

    Justification: We did not seek formal IRB review. Participants only reviewed code patches, much like their everyday code review work, and we collected no personal information. Participants were told that the patches came from automated coding systems (Appendix D.2). We consider the risk to participants minimal.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: LLMs are used as the agent backbones, as the pairwise judges, and across the data-curation pipeline. Models and configurations are documented in §4.1 and Appendix I.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.