CODEEVOGYM: LEARNING FROM EXPERIENCE ACROSS SOFTWARE RELEASES

Anonymous authors
Paper under double-blind review

ABSTRACT

Production web applications never stand still. Each release introduces features, fixes defects, and changes internal structure, making an agent's knowledge of the codebase increasingly outdated. This paper asks whether coding agents can continually update their understanding of a large web application as it evolves, and whether doing so improves their ability to implement changes in unseen releases. We introduce CODEEVOGYM, a benchmark and training environment constructed from the release histories of six production open-source web applications spanning three programming languages. Each task provides the source tree immediately preceding a real release, along with its associated development record: release notes, pull requests, issues, and their discussions. The agent's patch is evaluated using the tests added or modified in the post-release source tree. During training, the agent may query a grading oracle that executes the gold test suite in a private container and returns the number and identities of failing tests, together with an LLM-generated summary of their error patterns. The agent distills the resulting feedback into three persistent artifacts, a memory store containing application-specific knowledge, a skill library containing reusable procedures and documentation that is readable by both humans and LLM agents. These artifacts are frozen after the training releases and evaluated on subsequent unseen releases. We examine the impact of using these artifacts on agent performance and cost.

1 INTRODUCTION

Software is released, patched, and released again. Each release of a mature web application bundles fixes and small features, described in prose by its authors and ideally accompanied by tests of the new behaviour. A codebase's revision history is thus an ordered record of an evolving system, not a pile of independent changes. Yet most evaluations examine one change at a time. SWE-bench (Jimenez et al., 2024) set the unit as a single issue and its patch, solved in isolation by an agent starting from nothing. Later benchmarks enlarge the unit but keep it independent. Even SWE-EVO (Le et al., 2026), built from release transitions, treats each release as its own world. Continuity, the property a release sequence has and a pool of issues lacks, is discarded.

Continuity is how humans become experts on a codebase. By the tenth release, a maintainer knows where key functionality lives, which service must be up for the suite to run, how the project names things, and which subsystems break together. None of this is in the release notes; it is learned by making changes and watching the tests respond. An agent evaluated on independent issues never builds this knowledge, and one that does may fail to keep it as the code changes underneath it.

Practitioners already give agents persistent repository knowledge through context files such as `AGENTS.md`, skills, and long-term memory, with sobering results. Context files, whether written by developers or models, do not reliably improve success and raise inference cost by over a fifth (Gloaguen et al., 2026); self-written skills give no average benefit (Li et al., 2026a). Experience can compound for self-improving agents (Section 6), but only on streams of short, loosely related tasks. The closest attempt for code reorders an existing pool of issues by date (Joshi et al., 2025). None of this tests an agent on one large system that changes while it learns. We hypothesize that such knowledge is produced by reading code rather than changing it, written once, and never checked against execution. The open question is not whether an agent can store repository knowledge, but

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

whether it can *derive* it from feedback and *keep it current* as the code moves. We study this through three research questions.

RQ1 (Accumulation): can an agent evolve its own understanding of a large web application as the application evolves? Given real releases in chronological order and execution feedback on each attempt, does it build reusable knowledge of the codebase in artifacts that outlive the episode?

RQ2 (Transfer): does that self-evolution carry forward to unseen releases? With the learned artifacts frozen after the earlier releases, do they improve patch correctness on later releases of the same applications, compared with a stateless agent that uses the same model and tools?

RQ3 (Documentation): Can an agent document its own changes faithfully and legibly, so that the documentation is useful both to its future self and to humans? As agents write more code, such documentation may help keep human understanding of that code from degrading.

The agent harness does self-evolution through three persistent stores. A **memory** holds short declarative facts worth keeping across tasks. A **skill library** holds procedures the agent writes, names, and later retrieves and edits, such as how to bring the stack up or how to add a test fixture in the project's style. Additionally, we propose a **documentation store** holds prose that explains the code: an evolving description of the whole repository architecture, a *Journal* recording what each patch actually changed, and *Lessons* recording what execution feedback on each attempt taught the agent.

Answering the research questions above requires an environment in which a codebase genuinely moves and correctness is decided by execution rather than by similarity to a reference patch. Correspondingly we have built CodeEvoGym from six production open-source web applications spanning PHP, TypeScript, and Python. Each project's release history becomes an ordered sequence of tasks. The release notes are the specification, the containerized application at the preceding version is the starting state, and the tests introduced by that release define the target behaviour. Releases are split in time. The agent writes its memory, skills, and documentation while working through a project's earlier releases, and is then evaluated, with those artifacts frozen, on later releases it has never seen. After every attempt, a private oracle reports how many target tests now pass, how many previously passing tests broke, and an LLM-written summary of their shared failure patterns. This feedback is the only signal the agent receives, and the only material from which its artifacts can be built. The test code itself stays hidden, so the agent can learn the codebase but cannot learn the answer key.

Findings. Release-level evolution is hard even for strong models. The best stateless agent fully resolves only 19% of held-out releases (5 of 26), although none of its patches break previously passing tests, and it does far better on small releases than on large ones. Frozen artifacts do not change how many held-out releases are resolved, but they raise the micro fail-to-pass (F2P) rate (Section 2.3) by 20-46% relative across models, with gains coming mostly from large releases. They also lower cost by 15-25% across models, consistent with less effort spent on setup and localization. The quality of the artifacts matters as much as having them: with a stronger model's artifacts, a weaker model gets the same F2P gain for 25% less cost, but introduces regressions.

Our work makes the following contributions:

1. **A self-evolution benchmark over real codebase evolution:** tasks built from the release histories of six production open-source web apps across PHP, TypeScript, and Python, split in time so an agent learns on earlier releases and is tested on later, unseen ones.
2. **Execution feedback as the self-improvement signal:** a private grading oracle based on post-release tests (including unit tests, playwright tests, etc.) that returns progress and regression counts, the failing tests, and an LLM-written summary of their shared error patterns, and nothing else.
3. **A documentation store as a third self-evolution artifact:** alongside memory and skills, an architecture description of the repository, a per-task journal of what the patch changed, and per-task lessons drawn from execution feedback, each revised as the codebase and the agent's understanding of it evolve.
4. **An empirical study of accumulation, transfer, and self-documentation:** self-evolved artifacts raise micro F2P by 20-46% relative on unseen releases, and reduce cost by 15-25% and yield journal entries that faithfully describe the patch on 85 of 86 tasks.

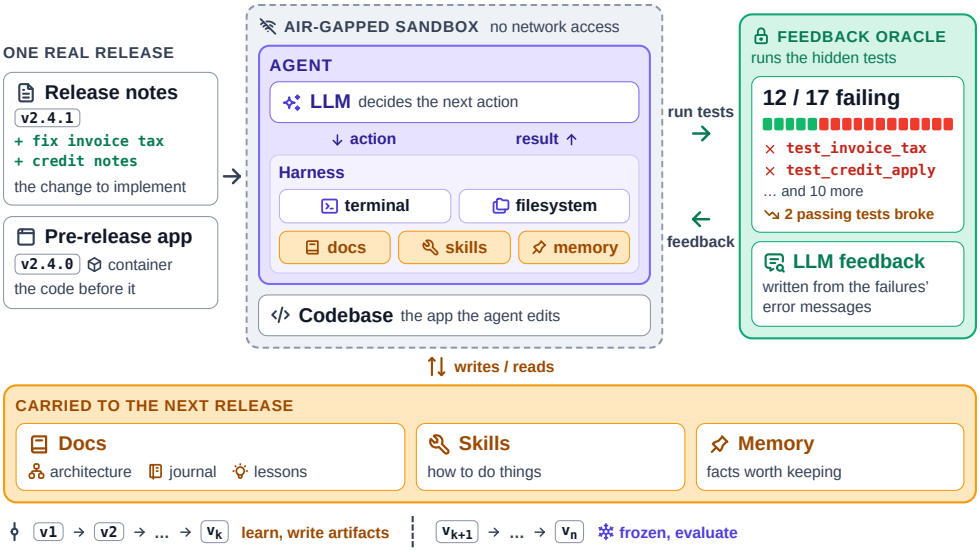


Figure 1: **Overview of CodeEvoGym.** CodeEvoGym turns the release history of an open-source web application into a sequence of tasks (§2). One task is one real release: its release notes and the application at the preceding version (left). The agent attempts the release inside an air-gapped sandbox (centre) and is graded by the release's own hidden tests, which return what now passes, what broke, and an LLM-written summary of the failures (right). What the agent learns about the codebase and the work, as docs, skills, and memory, is carried to the agent that attempts the next release (bottom). Releases are split in time: artifacts are written on earlier releases and frozen for evaluation on later ones.

2 CodeEvoGym: Task Formulation

CodeEvoGym transforms a sequential history of open-source web application releases into a sequence of agent tasks. As shown in Figure 1, each task is grounded in a real-world release, attempted by the agent within a sandboxed, air-gapped environment, and assessed against that release's own hidden test suite. The resulting knowledge of the codebase and of the work performed carries forward to inform the agent tackling the subsequent release.

2.1 Releases, Lines, and Windows

Let $\mathcal{R}$ be a corpus of repositories, each publishing releases $v_1, v_2, \ldots$ with publication times $t(v_i)$ and source trees $S(v_i)$. We order the releases based on their release version number (`7.10.3`). A **window** is a release v_i paired with the release v_{i-1} immediately before it. Windows need not be consecutive in the task sequence, since we keep only the release pairs with the most number of new tests created or edited in release v_i over v_{i-1}. A task pairs a release v_i with its predecessor v_{i-1}:

$$\tau = \left(S_\tau^{\text{pre}},\ S_\tau^{\text{post}},\ D_\tau,\ G_\tau\right),$$

where S_τ^{pre} and S_τ^{post} are the source trees immediately before $S(v_{i-1})$ and after the release $S(v_i)$, respectively, G_τ is the **gold test set** (the test files the release added or modified, at their post-release content), and D_τ is the description: the release notes, the pull requests, the issues those pull requests close, and the discussion threads on both.

The agent is given S_τ^{pre} and D_τ only ("pre-release app" and "release notes" in Figure 1); G_τ is never in its sandboxed container, and S_τ^{post} is used solely for grading.

2.2 The Agent and Its Environment

An agent π, composed of an LLM along with its agent harness, acts through a sandboxed environment E: a shell terminal and file editing tools inside a container holding S_τ^{pre} and the repository's

toolchain and services, and produces a patch $\Delta_\tau = \pi(S_\tau^{\text{pre}}, D_\tau, E)$. Let σ strip from Δ_τ any edits to paths that belong to the environment rather than the agent (test directories, runner configuration), so the agent may read the existing tests but cannot alter what it is graded against. The graded tree is $S_\tau^{\text{pre}} \oplus \sigma(\Delta_\tau)$.

Harness. An agent harness enables an LLM to be a coding agent that can interact with the environment through a set of tools, filesystem, and terminal, with the well known OpenHands system being a representative example Wang et al. (2025a). When the LLM chooses an action, the harness executes it in the container and returns the result. Other harnesses like Hermes[1] also support self-evolution through a self-curated persistent memory and a skill library using the Agent Skill format Anthropic (n.d.), that the agent can update. As described further in Section 2.6, we build on the Hermes harness in this work and propose three types of document stores that the agent can additionally use and update as it learns more about the codebases.

Sandbox air gap. Every task asks the agent to reconstruct a change that exists publicly upstream, so with network egress the cheapest available strategy is to implement nothing and fetch the reference diff instead. Hence, the agent sandbox is air-gapped (i.e. has no internet access) in our setting. Two paths are deliberately not air-gapped: the image build and the scoring container as dependencies are baked at build time, which is what makes runtime isolation possible.

2.3 The validity partition

Let $O(S, G)$ run the test suite on tree S with gold tests G seeded in, sorting every test into one of four outcomes: passed, failed, errored, or skipped. Skipped is its own outcome rather than a share of passed or failed, since a skip means the runner declined to answer.

Before any agent runs, we compute two such runs per release:

$$B_\tau = O(S_\tau^{\text{pre}}, G_\tau) \quad \text{(baseline)}, \qquad Q_\tau = O(S_\tau^{\text{post}}, G_\tau) \quad \text{(oracle)}.$$

Gold tests skipped in either run are excluded and the rest fall into two groups:

- **Fail-to-pass (F2P):** failed or errored in B_τ, but passed in Q_τ. This is the work the release actually does, and what the agent is asked to fix.
- **Pass-to-pass (P2P):** passed in both B_τ and Q_τ. This is what the agent must not break.

Every other gold test, one that failed, errored, or stayed unresolved in both runs, simply falls outside F2P and P2P and plays no further role in grading.

2.4 Scoring

Let $A_\tau = O(S_\tau^{\text{pre}} \oplus \sigma(\Delta), G_\tau)$ be the run on the agent's patched tree. Then:

$$\text{f2p_rate}(\tau) = \frac{\#\{\text{F2P tests passing in } A_\tau\}}{\#\{\text{F2P tests}\}}, \qquad \text{p2p_rate}(\tau) = \frac{\#\{\text{P2P tests passing in } A_\tau\}}{\#\{\text{P2P tests}\}}.$$

A task is marked **regressed** if at least one P2P test that passed before no longer passes in A_τ; otherwise it is not regressed. A task is marked **resolved** exactly when both conditions hold: the F2P rate equals 1, and the task is not regressed. (Equivalently, resolved is the logical AND of "f2p_rate $= 1$" and "not regressed" - we state it this way in place of an indicator-function expression, since it is just these two boolean conditions combined.)

The F2P rate is what gives partial credit - it separates an agent that did most of the work from one that did none, which a pass/fail verdict cannot do. If a task has no F2P tests (or no P2P tests), the corresponding rate is reported as undefined rather than 0.

The headline **resolved rate** is

$$\text{resolved}_{\text{valid}} = \frac{\sum_{\tau \in \mathcal{T}_{\text{valid}}} \text{resolved}(\tau)}{|\mathcal{T}_{\text{valid}}|},$$

where resolved(τ) is taken to be 1 if τ is resolved and 0 otherwise, and $\mathcal{T}_{\text{valid}}$ excludes tasks whose environment never ran.

[1] https://github.com/NousResearch/hermes-agent

2.5 TRAIN AND EVAL SPLITS, AND THE FEEDBACK ORACLE

For each repository, a temporal cutoff c is set at the point 70% of the way through its task timeline, splitting tasks into $\mathcal{T}_{\text{train}} = \{\tau : t(v_i) \le c\}$ and $\mathcal{T}_{\text{eval}}$, so every eval task comes from a later release than anything the agent learned from.

Train tasks expose a **feedback oracle** Φ_τ, which the agent may invoke at will (Figure 1, right):

$$\Phi_\tau(\Delta) = \big(\#\text{F2P passing}, \#\text{F2P total}, \#\text{P2P passing}, \#\text{P2P total}\big),$$

along with the names of still-failing F2P_τ and regressed P2P_τ tests and a short excerpt of the error message, as a developer reading CI would see. Φ_τ is answered by a private grading container that applies the diff exactly as scoring does, so G_τ never enters the agent's container. A separate LLM call to another instance of the same LLM that is being tested also reads that check-in's failures and returns a few sentences on the *pattern* behind them, without proposing code or naming files, verbal feedback in the spirit of Reflexion (Shinn et al., 2023) and reinforcement learning with execution feedback (Gehring et al., 2025). Eval tasks receive no feedback: Φ_τ is undefined and the patch is graded once, after the episode ends.

2.6 SELF-EVOLUTION: CARRYING THE RELEASE FORWARD

A developer who has shipped ten releases of a codebase is faster on the eleventh because of what they carry, not because they reason better: the next release touches the same modules, the same fragile fixtures, the same tooling. LLM agents show the same effect when given somewhere to put experience, for example, stored self-reflection (Shinn et al., 2023), skill libraries (Wang et al., 2024a), memory with reflection (Park et al., 2023; Packer et al., 2023), and distilled insights and workflows (Zhao et al., 2024; Wang et al., 2024b).

The cycle. Let $M = (M^{\text{doc}}, M^{\text{skill}}, M^{\text{mem}})$ be the persistent store, initially empty (no bundled skills: everything in it was written by an agent). Train tasks $\tau_1, \ldots, \tau_n$ run in release order, one at a time, each reading the store left by all earlier ones:

$$\big(\Delta_{\tau_j},\, M_j\big) = \pi\big(S^{\text{pre}}_{\tau_j},\, D_{\tau_j},\, E,\, \Phi_{\tau_j},\, M_{j-1}\big), \qquad M_0 = \emptyset. \tag{1}$$

Eval tasks run without Φ against a copy of M_n in which skills and memory are frozen and only documentation is writable, so eval agents inherit everything train learned but cannot change the procedure later eval agents inherit.

Three stores. As seen in Figure 1 (bottom), **Skills** (M^{skill}, Hermes') hold *how to do things*: markdown playbooks, including one per repository with the commands that actually run its tests and the approaches that wasted earlier agents' time. **Memory** (M^{mem}, Hermes') holds short facts worth keeping that are *not* specific to any repository, loaded into every later system prompt. **Docs** (M^{doc}, added by CodeEvoGym) hold *what the codebase is*: an architecture map, a journal with one entry per release attempted (what changed, what Φ reported), and a lessons document on what the graded suite revealed. The prompt separates docs from skills by one rule: a sentence that describes the code is documentation; one that says what to do is a skill.

Why documentation is a separate store. Neither Hermes store fits a structural account of one codebase, which is what a returning developer relies on most. M^{doc} is written by the agents that had to work the code out, release by release, and checked against what Φ reported. Additionally, we also build three tools for the agent to interact with the docs: `doc_list` returns a cheap index, `doc_view` one document in full, and `doc_manage` writes (append-only for the journal and lessons, so concurrent writers cannot collide).

3 BUILDING THE GYM

CodeEvoGym is built in four stages in order:

Stage 1: the release scraper. The scraper turns a repository's release history into a structured on-disk dataset. Two gates apply to this scrape: a release is stored only if its comparison diff changes at least one test file, since a release with no test material can never be graded and, only the top K releases per repository are downloaded, ranked by test material.

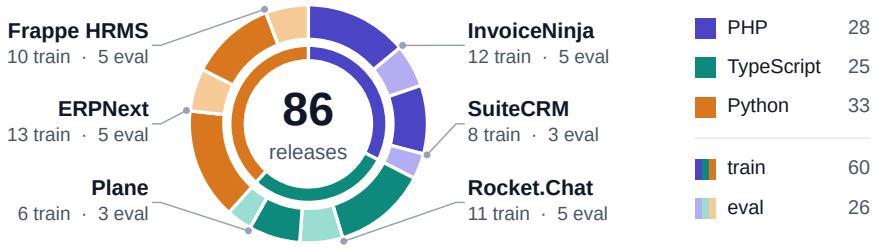


Figure 2: **Composition of CodeEvoGym.** Six production web applications across three stacks (inner ring), each split in time into train and evaluation releases (outer ring; lighter shade is evaluation), for 60 train and 26 evaluation releases in total. A train task changes 12,299 lines of code on average between S^{pre} and S^{post}, and an evaluation task changes 26,040, roughly twice as many.

Stage 2: the task builder. The task builder turns a (repository, window) pair into a task episode and assembles D_τ and G_τ. For the description D_τ, each release contributes its notes prose with the leading metadata block stripped, then, the pull requests behind it paired with the issues they close so that the two read as one item. Comment threads are also added, with bot comments dropped.

Stage 3: the oracle computation. Which gold tests a release can measure is a property of the scraped repository, not of any particular run, so the baseline and oracle are computed deliberately, once per release, and stored inside that release's own directory.

Stage 4: export, containers, and the prompt Export builds one container image and one prompt per task, plus a sidecar file carrying the task identity, gold set, split, and baseline/oracle identifier lists needed to reconstruct and score the task later. The image build has two layers. A per-repository **environment image** installs the source-independent toolchain and service binaries once and is reused across every task of that repository. Each **task image** is built on top of it, copies the pre-release source, bakes the project dependencies, and creates a baseline git commit so the agent's own edits are recoverable as a diff. The **prompt** along with the filesystem are the only channels into a run, so everything the agent needs is embedded in either the prompt or the disk. The prompt states the source location, the release description, pointers to the pull request and issue material in the filesystem that the agent can access, and an explicit tool-call budget.

4 Experimental Setup

4.1 Dataset Composition

The corpus is drawn from production-grade, self-hostable open-source enterprise web applications: the class of software that is deployed, operated, and maintained over years, as opposed to the libraries that dominate existing benchmarks. As seen in Figure 2, CodeEvoGym has a catalogue 86 release tasks (60 train, 26 eval) from 6 applications[2] (InvoiceNinja, SuiteCRM, Rocket.Chat, Plane, ERPNext, Frappe HRMS) spanning ERP (Enterprise Resource Planning), CRM (Customer Relationship Management), project management, HR (human resources), accounting, etc.

4.2 Models and Settings Evaluated

We evaluate four models as the policy π: GPT 5.6 Sol, GPT 5.6 Luna, Claude Opus 5.5, and Claude Sonnet 5. For each model we report a stateless baseline and a self-evolution condition, each with a budget of 150 LLM API calls per task. In the stateless baseline, the model attempts every task in T_{eval} on its own, with the persistent store held at $M = \emptyset$ throughout: the agent receives only S^{pre}_τ, D_τ, and the tools in E, with no memory, skills, or documentation carried over from any other episode. This is the stateless agent referenced in RQ2, and it measures what each model can do from the task description alone, with no opportunity to accumulate codebase-specific knowledge. In the self-evolution condition, the model runs the full cycle described in Section 2.6: it works

[2] invoiceninja/invoiceninja, SuiteCRM/SuiteCRM, rocketchat/rocket.chat, makeplane/plane, frappe/erpnext, frappe/hrms

Model	Self-Evo.	#Resolved	F2P Rate		P2P Rate		Run Cost
			macro	micro	macro	micro	
GPT 5.6 Sol	No	3/26	35.6%	12.3%	100%	100%	$86.51
	Yes	3/26	35.3%	17.9%	100%	100%	$71.86
GPT 5.6 Luna	No	2/26	20.0%	6.0%	99.9%	99.9%	$2.00
	Yes (Luna Artifacts)	2/26	19.0%	8.3%	100%	99.8%	$2.09
	Yes (Sol Artifacts)	2/26	19.3%	8.3%	96.3%	96.8%	$1.50
Claude Opus 5.5	No	5/26	44.3%	21.3%	100%	100%	$42.18
	Yes	5/26	46.0%	25.6%	100%	100%	$35.88
Claude Sonnet 5	No	3/26	30.4%	10.3%	100%	99.9%	$19.78
	Yes	3/26	30.0%	13.7%	100%	99.9%	$17.12

Table 1: Resolve rate, F2P/P2P rates, and total run cost by model and self-evolution setting, discussed further in Section 5.

through T_{train} in release order, reading and writing M^{doc}, M^{skill}, and M^{mem} as it goes and consulting the feedback oracle Φ_τ where available, and the resulting store M_n is frozen and carried into T_{eval}. Additionally, we report two self-evolution conditions for Luna. In the first, Luna Artifacts, Luna follows the same procedure as the other models: it runs its own training cycle over T_{train} and is evaluated on T_{eval} against the frozen store M_n it wrote itself. In the second, Sol Artifacts, Luna is instead evaluated on T_{eval} against the frozen store M_n produced by GPT 5.6 Sol's own training run: Luna never executes T_{train} itself, and instead inherits Sol's memory, skills, and documentation as a fixed, read-only context at evaluation time. Comparing these two conditions isolates the effect of artifact quality, written by a different and more capable model.

5 Results and analysis

Self-evolution improves partial progress, not resolution. As seen in Table 1, across all three models, self-evolution increases micro F2P (46% relative for Sol, 38% for Luna, 20% for Opus) but does not change the number of resolved releases. Artifacts increase how much of a release agents implement, but not whether they implement all of it. The gain does not come at the expense of regressions: P2P stays intact for Sol and Opus.

Gains concentrate in large releases. Macro F2P is flat while micro F2P rises. This is because newly passing tests mainly come from the more complex releases with many F2P tests. These are the releases where a stateless agent struggles most, and where knowing the codebase matters more than reading the release notes.

The effect is application-dependent. Figure 3 breaks Sol's micro F2P down by application. The gains are largest on Plane (12.82%→33.97%, +21.2 points), then SuiteCRM (+7.7) and Frappe HRMS (+5.6). InvoiceNinja stays at 1.95%, near the floor in both conditions. Self-evolution hurts on ERPNext (-4.0) and Rocket.Chat (-6.6). Plane accounts for more newly passing tests than the overall net gain. This is because it has a task with 136 F2P tests, as compared to 20 tests on average in the overall evaluation set, which the self-evolution equipped Sol is able improve.

Knowing the codebase makes agents cheaper, if the artifacts are good. Self-evolution cuts Sol's cost by 17% and Opus by 15%. This is consistent with an agent that spends less effort on setup and localization and reaches the limit of what it can implement sooner. Luna shows that the saving depends on who wrote the artifacts. With its own store, Luna's cost increases slightly, since reading the store roughly offsets the exploration it saves. With Sol's store, Luna's cost falls by 25% at the same F2P rate. The benefit therefore comes from the quality of the stored knowledge, not from having a store at all. Sol's artifacts also introduce the only notable regressions (lower P2P rate) in Table 1. We hypothesize that a weaker model follows a stronger model's procedures without the understanding that made them safe to apply, acting sooner and verifying less.

What agents write. Documentation makes up almost 90% of the models' stores (Table 2). Sol writes 1.4-1.9× more than Luna in every store except memory, which the Hermes harness caps at

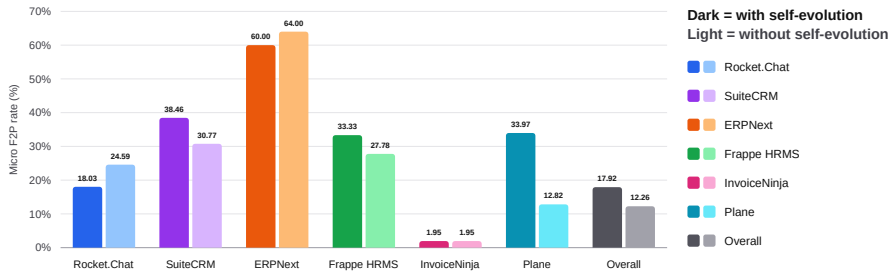


Figure 3: **Repo-wise micro F2P rate for GPT 5.6 Sol on CodeEvoGym, with and without self-evolution.** Enabling self-evolution improves the micro F2P rate on Frappe HRMS, SuiteCRM, and Plane, leaves InvoiceNinja unchanged, and slightly reduces performance on ERPNext and Rocket.Chat, yielding a net improvement in the Overall rate (17.92% vs. 12.26%).

Model	Memory	Skills	Documents		
			Architecture	Journal	Lessons
GPT 5.6 Sol	2198	7960	28141	36279	19712
GPT 5.6 Luna	2070	5435	14750	21598	10354
Claude Opus 5.5	2157	7456	26329	37302	18178
Claude Sonnet 5	2120	6342	19320	30768	14283

Table 2: Average number of characters in various self-evolution artifacts created by each model.

2,200 characters and which the models nearly fill. Sol's larger store coincides with its lower cost, so reading more does not undo the savings. To answer **RQ3**, we had Claude Opus 5.5 act as a judge for Sol's run. For each task, the judge read the agent's trajectory, its final patch, and the journal entry it wrote, and decided whether the entry faithfully described the change. The journal was accurate on 85 of 86 tasks. The one failure occurred at evaluation time, when an agent claimed that two tests named in the release already passed on the pre-release code without having read the relevant files. Without the feedback oracle, nothing could catch the claim. Agents therefore document their own changes faithfully, and failure comes from an unverified assertion rather than a misdescribed patch.

6 Related Work

From single issues to software evolution. SWE-bench evaluates repository-level coding by pairing real GitHub issues with the tests that resolve them (Jimenez et al., 2024). Later work carries this setting to more languages and visual domains (Yang et al., 2025; Zan et al., 2025), keeps it fresh against contamination (Badertdinov et al., 2025; Zhang et al., 2025), audits its reliability (Aleithan et al., 2024), and scales it to long, multi-file changes (Deng et al., 2026b; Yang et al., 2026). Agent scaffolds such as SWE-agent and OpenHands show that the harness around the model strongly affects results (Yang et al., 2024; Wang et al., 2025a). More recent benchmarks evaluate sequences of changes. SWE-EVO (Le et al., 2026) builds release-sized tasks from release notes, but its 48 Python-library instances are evaluated independently, with no train split and nothing carried between releases. CodeEvoGym instead orders the releases of six complex multi-language web applications in time, gives execution feedback on the earlier ones, and freezes the agent's self-written artifacts before evaluating on later ones. SWE-EVO asks how hard a release is, while CodeEvoGym asks whether an agent gets better at a codebase across releases. SWE-Chain and EvoClaw chain releases or milestones so that each task starts from the agent's own previous output, and find that errors compound along the chain (Lam et al., 2026; Deng et al., 2026a). SWE-CI asks whether correctness survives later modifications (Chen et al., 2026). In all of these, what carries from one task to the next is *code*. In CodeEvoGym, every release starts from the true pre-release state, so what carries forward is *knowledge*: the agent's memory, skills, and documentation. This separates learning from error propagation. SWE-Bench-CL, the closest prior work, orders issues by date to study continual

learning (Joshi et al., 2025). CodeEvoGym differs in its unit of work (whole releases of production web applications), its signal (a private execution oracle), and its protocol (artifacts frozen before evaluation).

Memory, skills, and self-evolving agents. Agents can improve across tasks without changing their weights. Reflexion turns failures into verbal lessons (Shinn et al., 2023). ExpeL distils insights from past trajectories (Zhao et al., 2024). Voyager and Agent Workflow Memory store reusable skills and procedures (Wang et al., 2024a; 2025b). Generative agents, MemGPT, and evolving-context methods manage long-lived memory (Park et al., 2023; Packer et al., 2023; Suzgun et al., 2025; Zhang et al., 2026b). MemSkill and CODESKILL learn and refine skills for general and coding agents (Zhang et al., 2026a; Li et al., 2026b). Other work improves the harness itself (Lee et al., 2026; Karten et al., 2026), and a recent survey sorts self-evolving coding agents by what they change (Zhou et al., 2026). Evo-Memory benchmarks this ability on streams of short tasks (Wei et al., 2025a). For coding agents, repository context files such as `AGENTS.md` rarely improve success (Gloaguen et al., 2026), and self-generated skills give no average benefit (Li et al., 2026a). In all of these settings the environment stays fixed between tasks. A codebase does not: a fact about it can go stale after one release, and a procedure that worked on one version can break on the next. CodeEvoGym measures whether stored knowledge remains useful under such change, and adds documentation, prose that explains the code, as a third artifact beside facts and procedures.

Execution feedback as supervision. Software benchmarks judge correctness by executing code rather than comparing it with a reference patch. Execution signals are also used to train code models directly (Gehring et al., 2025) and to learn from open-source software evolution (Wei et al., 2025b). Test success can still overstate quality: SWE-Gate shows that patches can pass functional tests while violating constraints raised in code review (He et al., 2026). TestEvo-Bench studies the reverse problem, evolving tests alongside code (Wang et al., 2026). CodeEvoGym uses each release's own tests as hidden evidence. Fail-to-pass tests capture the behaviour the release introduced, and pass-to-pass tests catch regressions. The agent sees counts, failing test names, and a summary of shared errors, but never the tests themselves or the post-release source. The agent can act on this feedback without seeing the reference solution. Our temporal split follows time-consistent repository evaluation (Sun et al., 2026): the agent learns only from releases that precede the ones it is evaluated on.

Realistic environments and harnesses. Agent performance depends on the environment and harness, not only on the model. Terminal-Bench evaluates long-horizon terminal work with executable checks (Merrill et al., 2026), MCPVerse tests tool use at large action scale (Lei et al., 2026b), DA-Comp targets enterprise data engineering (Lei et al., 2026a), and JourneyBench targets policy-aware support workflows (Balaji et al., 2026). HAL shows that comparisons between agents are sensitive to harness details (Kapoor et al., 2026). Web applications are harder still, because their behaviour is spread across interfaces, APIs, databases, and background services. SWE-bench Multimodal and ProgramDistill study visual repair and application reconstruction (Yang et al., 2025; Kim et al., 2026). CodeEvoGym instead extends an existing application release by release. It ships each application as a container with a fixed toolset, budget, feedback protocol, and artifact interface, so results are comparable across agents.

7 Conclusion

We introduced CodeEvoGym, an environment which tests whether an agent can learn a large web application from execution feedback, keep that knowledge current as the code changes, and eventually ship full releases on its own. It turns the release history of production web applications into an ordered sequence of tasks, so that an agent can learn a codebase on earlier releases and be tested, with its knowledge frozen, on later ones it has never seen. The only learning signal is execution feedback from a private oracle, and what the agent keeps is memory, skills, and documentation it wrote itself. Our experiments suggest that this kind of accumulated knowledge is real but modest in its effect: it helps agents make more progress on large releases and reduces the effort spent rediscovering how a project is built and tested, without changing whether a release is fully resolved. The value of the store depends on who wrote it, and inheriting a stronger model's artifacts trades cost for regressions. Agents can also document their changes quite faithfully, which may prove to be essential to help humans keep up as machines write more code.

AI USE STATEMENT

Generative AI tools were used to support paragraph-level language editing, improve the organization and coherence of the introduction, identify relevant literature, and assist with code development and revision. They were also used to prepare preliminary drafts of selected sections; these drafts were subsequently rewritten and replaced substantially by the authors.

Generative AI was not used to develop the research ideas. Other disclosure categories, including the generation of synthetic datasets and the verification of mathematical claims, are not applicable to this work. All AI-assisted material was reviewed by the authors. In particular, AI-generated code was inspected, and the related-work suggestions were independently read and incorporated by the authors. The authors assume full responsibility for the final manuscript and for all text, claims, code, and other artifacts produced with assistance from generative AI tools.

LIMITATIONS

Six applications and 26 held-out releases for evaluation are enough to observe consistent effects across models, but per-application results rest on a few tasks each, and a single release with many tests can shape the aggregate F2P rate, as Plane does here. The scraper keeps the most test-rich releases of each repository, which makes tasks gradable but means the corpus favours releases with substantial test changes over routine ones. As in other execution-based benchmarks, a release's tests define correctness, and passing them does not guarantee the patch is one a maintainer would merge. The training oracle and the RQ3 faithfulness check both involve an LLM, the former to summarise failure patterns and the latter as a judge. We constrain the oracle to name no files and propose no code, but neither signal is purely mechanical. All six repositories are public, so the models may have encountered their later versions during pretraining. The air gap rules out retrieval at run time but not prior exposure, and our temporal split mitigates rather than eliminates this concern. Finally, results are specific to one harness and one artifact interface. Other designs for memory, skills, and documentation may shift where the gains appear, and we release the environment so that they can be tested under the same protocol.

REFERENCES

Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. Swe-bench+: Enhanced coding benchmark for llms. *arXiv preprint arXiv:2410.06992*, 2024.

Anthropic. Extend claude with skills – claude code docs. Anthropic documentation, n.d. URL https://docs.anthropic.com/en/docs/claude-code/skills. Accessed: 2026-05-31.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents, 2025. URL https://arxiv.org/abs/2505.20411.

Sumanth Balaji, Piyush Mishra, Aashraya Sachdeva, and Suraj Agrawal. Beyond IVR: Benchmarking customer support LLM agents for business-adherence. *arXiv preprint arXiv:2601.00596*, 2026. URL https://arxiv.org/abs/2601.00596.

Jialong Chen, Xander Xu, Hu Wei, Chuan Chen, and Bing Zhao. Swe-ci: Evaluating agent capabilities in maintaining codebases via continuous integration. *arXiv preprint arXiv:2603.03823*, 2026.

Gangda Deng, Zhaoling Chen, Zhongming Yu, Haoyang Fan, Yuhong Liu, Yuxin Yang, Dhruv Parikh, Rajgopal Kannan, Le Cong, Mengdi Wang, et al. Evoclaw: Evaluating ai agents on continuous software evolution. *arXiv preprint arXiv:2603.13428*, 2026a.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Y. He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa R Kundurthy, Sean M. Hendryx,

Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-bench pro: Can AI agents solve long-horizon software engineering tasks?, 2026b. URL https://openreview.net/forum?id=9R2iUHhVfr.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. RLEF: Grounding code LLMs in execution feedback with reinforcement learning. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=PzSG5nKe1q.

Thibaud Gloaguen, Niels Mündler, Mark Niklas Müller, Veselin Raychev, and Martin Vechev. Evaluating AGENTS.md: Are repository-level context files helpful for coding agents? *arXiv preprint arXiv:2602.11988*, 2026.

Xin He, Yanlin Wang, Mingwei Liu, Jiachi Chen, Hongyu Zhang, and Guanbin Li. Swegate: Passing functional tests is not enough for software engineering agents. *arXiv preprint arXiv:2609.04167*, 2026.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Thomas Joshi et al. SWE-bench-CL: Continual learning for coding agents. *arXiv preprint arXiv:2507.00014*, 2025.

Sayash Kapoor, Benedikt Stroebl, Peter Kirgis, Nitya Nadgir, Zachary S. Siegel, Boyi Wei, Tianci Xue, Ziru Chen, Felix Chen, Saiteja Utpala, Franck Ndzomga, Dheeraj Oruganty, Sophie Luskin, Kangheng Liu, Botao Yu, Amit Arora, Dongyoon Hahm, Harsh Trivedi, Huan Sun, Juyong Lee, et al. Holistic agent leaderboard: The missing infrastructure for ai agent evaluation. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=vUaY1t64ZZ.

Seth Karten, Joel Zhang, Tersoo Upaa Jr, Ruirong Feng, Wenzhe Li, Chengshuai Shi, Chi Jin, and Kiran Vodrahalli. Continual harness: Online adaptation for self-improving foundation agents. *arXiv preprint arXiv:2605.09998*, 2026.

Jeonghye Kim, Minseon Kim, Young Jin Kim, Matheus Pereira, Marc-Alexandre Côté, Alessandro Sordoni, Xingdi Yuan, and Zhengyan Shi. Programdistill: From interactive web apps to verifiable reference-guided swe tasks. *arXiv preprint arXiv:2609.18805*, 2026.

Man Ho Lam, Chaozheng Wang, Hange Liu, Jingyu Xiao, Haau-sing Li, Jen-tse Huang, Terry Yue Zhuo, and Michael R Lyu. Swe-chain: Benchmarking coding agents on chained release-level package upgrades. *arXiv preprint arXiv:2605.14415*, 2026.

Tue Le, Minh V. T. Thai, Dung Nguyen Manh, Huy Phan Nhat, and Nghi D. Q. Bui. Swe-evo: Benchmarking coding agents in long-horizon software evolution scenarios, 2026. URL https://arxiv.org/abs/2512.18470.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

Fangyu Lei, Jinxiang Meng, Yiming Huang, Junjie Zhao, Yitong Zhang, Jianwen Luo, Xin Zou, Ruiyi Yang, Wenbo Shi, Yan Gao, Shizhu He, Jun Zhao, Zuo Wang, Qian Liu, Yang Wang, Wang Ke, and Kang Liu. Dacomp: Benchmarking data agents across the full data intelligence lifecycle. In *International Conference on Learning Representations*, 2026a. URL https://openreview.net/forum?id=EtzJy9yI5J.

Fei Lei, Yibo Yang, Wenxiu Sun, and Dahua Lin. Mcpverse: An expansive, real-world benchmark for agentic tool use. OpenReview submission, 2026b. URL https://openreview.net/forum?id=BjZvLXvLZW.

Xiangyi Li, Wenbo Chen, Yimin Liu, Shenghan Zheng, Xiaokun Chen, Yifeng He, Yubo Li, Bingran You, Haotian Shen, Jiankai Sun, et al. SkillsBench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026a.

Yanzhou Li, Yiran Zhang, Xiaoyu Zhang, Xiaoxia Liu, and Yang Liu. Codeskill: Learning self-evolving skills for coding agents. *arXiv preprint arXiv:2605.25430*, 2026b.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026. URL https://arxiv.org/abs/2601.11868.

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

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pp. 1–22, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Haonan Sun, Tian Yu, Sheng Ma, Qincheng Zhang, Lifei Rao, Chen Tian, et al. Atime-consistent benchmark for repository-level software engineering evaluation. *arXiv preprint arXiv:2603.26137*, 2026.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory, 2025. URL https://arxiv.org/abs/2504.07952.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024a. ISSN 2835-8856. URL https://openreview.net/forum?id=ehfRiF0R3a.

Jiale Amber Wang, Kaiyuan Wang, and Pengyu Nie. Testevo-bench: An executable and live benchmark for test and code co-evolution. *arXiv preprint arXiv:2607.02469*, 2026.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025a.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024b.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Forty-second International Conference on Machine Learning*, 2025b. URL https://openreview.net/forum?id=NTAhi2JEEE.

Tianxin Wei, Noveen Sachdeva, Benjamin Coleman, Zhankui He, Yuanchen Bei, Xuying Ning, Mengting Ai, Yunzhe Li, Jingrui He, Ed H Chi, et al. Evo-memory: Benchmarking llm agent test-time learning with self-evolving memory. *arXiv preprint arXiv:2511.20857*, 2025a.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *Advances in Neural Information Processing Systems*, 38:78500–78525, 2025b.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=mXpq6ut8J3.

John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, Diyi Yang, Sida Wang, and Ofir Press. SWE-bench multimodal: Do AI systems generalize to visual software domains? In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=riTiq3i21b.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. Program-bench: Can language models rebuild programs from scratch?, 2026. URL https://arxiv.org/abs/2605.03546.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-swe-bench: A multilingual benchmark for issue resolving, 2025. URL https://arxiv.org/abs/2504.02605.

Haozhen Zhang, Quanyu Long, Jianzhu Bao, Tao Feng, Weizhi Zhang, Haodong Yue, and Wenya Wang. Memskill: Learning and evolving memory skills for self-evolving agents. *arXiv preprint arXiv:2602.02474*, 2026a.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. Swe-bench goes live!, 2025. URL https://arxiv.org/abs/2505.23419.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models, 2026b. URL https://arxiv.org/abs/2510.04618.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Hao Zhou, Haichuan Hu, Tianyu Luo, Ye Shang, Chunrong Fang, Zhenyu Chen, Liang Xiao, and Quanjun Zhang. Self-evolving coding agents, 2026. URL https://arxiv.org/abs/2608.03392.