

Harness Self-Evolution: Theory, Algorithms, Evaluation, Applications, and Impact

Chuyang Wei^{1,3}, Haoxiang Guan^{1,3}, Minghang Zhu¹, Kexin Zhou^{1,3},
Jiyan He^{1,2}, Xiren Zhou³, Huanhuan Chen³, Yifei Shen^{1,2,†}

¹Zhongguancun Academy · ²Zhongguancun Institute of Artificial Intelligence

³University of Science and Technology of China · [†]Corresponding author

Abstract

Building agents that learn from experience is a central goal of artificial intelligence. Harness self-evolution offers a path toward this goal by adapting the system around a fixed base model. This article develops a unified framework for examining how execution feedback produces persistent harness changes. We organize algorithms by their update targets, distinguishing instructions, memory, and workflow specifications interpreted by fixed components from executable tools and runtime logic, and compare how they propose and admit changes. The theoretical analysis views harness evolution within the framework of evolutionary algorithms and draws on PAC learning theory for continual learning to provide insights into algorithm design. For evaluation, we distinguish four claims behind a reported gain: selection, whether the estimate is independent of the search or accounts for selection; activation, whether the accepted update operated in the scored runs; attribution, whether it caused the difference under controlled evaluation conditions; and improvement, whether the difference holds beyond the tasks that produced it. The reviewed protocols show why held-out performance, component use, and retained gains require different evidence. This article covers applications in personal assistance, enterprise automation, scientific discovery, robotics, and public services. We also discuss potential effects on productivity and skills, economic distribution and market structure, innovation and public services, privacy and security, and institutional governance. Drawing on these analyses, we identify future directions in updating algorithms, supporting infrastructure, evaluation, and responsible deployment.

✉ yshenaw@connect.ust.hk

🐱 <https://github.com/zgcagi/Harness-RSI>



The Path of Harness Self-Evolution

🚀 Harness evolution progresses from refining instructions to reshaping the systems that execute them

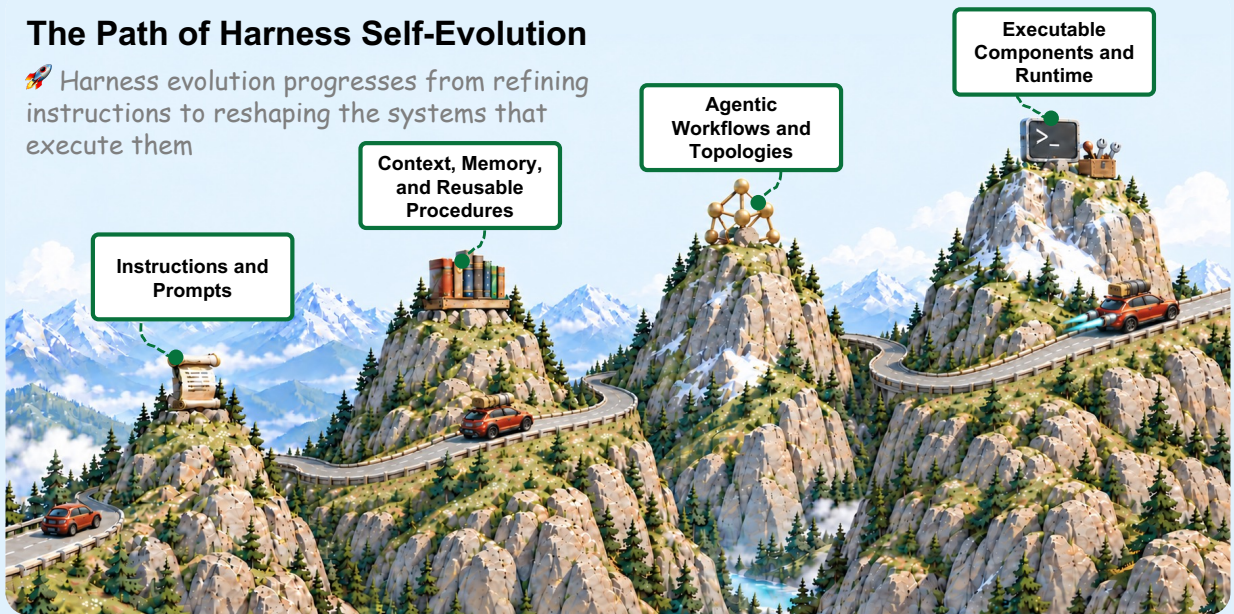


Figure 1 The path of harness self-evolution.

Contents

1	Introduction	3
2	Definition of Harness Self-Evolution	5
3	Interpreted Update Targets	7
3.1	The Rise of Harness RSI	9
3.2	Instructions and Prompts	9
3.3	Context, Memory, and Reusable Procedures	10
3.4	Agentic Workflows and Topologies	12
4	Executable Update Targets	12
4.1	Tools and Code Patches	13
4.2	Repository and Runtime Logic	14
4.3	Everything Is a Plugin	15
5	Theory of Harness Self-Evolution	15
5.1	An Evolutionary Perspective on Harness Search	16
5.2	Generalization and Retention from a Continual PAC-Learning Perspective	17
6	Evaluating Harness Self-Evolution	20
6.1	Four Claims Behind a Task Score	21
6.2	Current Evaluation Practices and Benchmarks	21
6.3	Longitudinal and Dynamic Evaluation	29
7	Applications of Harness Self-Evolution	32
7.1	Personal Assistance and Developer Workspaces	32
7.2	Enterprise Automation	33
7.3	Scientific Discovery	33
7.4	Robotics and Embodied Interaction	34
7.5	Public Services and Clinical Systems	34
8	Societal and Economic Implications	35
8.1	Productivity, Work, and Education	36
8.2	Economic Distribution and Market Structure	36
8.3	Innovation and Public-Service Capacity	37
8.4	Privacy, Security, and Systemic Stability	38
8.5	Governance and Institutional Power	38
9	Future Directions	39
9.1	Model–Harness–Environment Co-Evolution	39
9.2	Formal Representations for Harness Evolution	40
9.3	Synthesis and Reuse of Harness Plugins	41
9.4	Distributed Execution and Edge Infrastructure	41
9.5	Dynamic Evaluation and Benchmark Refresh	42
9.6	Safety Boundaries for Self-Modifying Systems	43
9.7	Governance, Provenance, and Human Oversight	43
10	Conclusion	44

1 Introduction

Progress toward artificial general intelligence (AGI) requires agents whose competence can develop after deployment. In open environments, agents face unfamiliar problems and changing task conditions. A fixed model can generalize to new problems, but repeated execution alone does not turn feedback into persistent revisions of instructions, memory, or control logic. Self-evolution makes these procedures revisable through experience, connecting early work on recursive self-improvement and self-referential problem solving with current self-evolving agents [1–4]. Revision can also change how experience is generated: tool choices and search paths shape which observations and failures feed later updates. Execution experience can therefore be history-dependent and non-i.i.d. even when tasks are sampled from a fixed distribution. The challenge is to make this feedback yield capabilities that remain useful across later tasks and revisions.

Self-evolution can target either the base model or its execution harness. Model updates alter internal weights; the harness is the external runtime that turns a general-purpose model into an operating agent by orchestrating prompts, memory, workflows, and executable tools across runs. We focus on harness self-evolution for three reasons. First, harness design strongly affects deployed behavior: performance varies substantially across scaffolds even when the model is fixed [5–7]. Second, it can operate without access to model weights, making adaptation possible with black-box, proprietary, or API-hosted models. Third, harness artifacts expose a readable and locally editable surface: instructions, memory entries, and code changes can be inspected, versioned, and compared. Our analysis concerns harness changes under a fixed base model; for co-evolving systems, we consider comparisons that isolate the harness contribution.

Recent work spans revisions to guidance and memory as well as workflows and executable components [6–10]. Figure 2 traces this growth and classifies representative systems by primary emphasis. The literature describes evolution through different update targets, feedback signals, and search procedures, making methods difficult to compare on a common basis. We organize the literature by the *update target*, what a proposal modifies, because it identifies the state that later runs inherit. Within each target, we compare how execution feedback produces revisions and how candidates are admitted.

We study harness self-evolution as a continuing operational feedback loop: the system’s own execution feedback drives revisions to harness state, and later runs inherit and load the accepted state (Section 2). We also review the offline optimizers these systems build on and related methods whose updates do not persist. Our taxonomy separates instructions, memory, and workflow specifications interpreted by fixed host components from executable implementations, including callable tools, harness source, and plugin compositions (Sections 3–4).

A controlled final-state comparison can establish an overall gain without showing what each revision contributed or whether the gain survives later updates. We distinguish four claims behind a reported gain: *selection*, whether estimation is independent of search or accounts for its influence; *activation*, whether the update operated in scored runs; *attribution*, whether it caused the observed difference; and *improvement*, whether the difference extends beyond the tasks that produced it. Section 6 develops these evidence criteria and uses them to examine current evaluation practices.

Existing surveys organize self-evolution by operational component [3], optimizer feedback [4], externalized infrastructure [11], the role of code [12], or update target paired with signal origin or loop closure [13, 14]. Table 1 compares their harness-relevant coverage. Recent surveys also address evaluation. Ren et al. [13] call for separate test data, independent final judges, and component substitution. Zhou et al.’s coding-agent survey [15] distinguishes runtime activation from candidate admission and proposes comparisons under a common budget with cost and activation recorded. We build on this work by applying comparison-level evidence criteria to reviewed results and examining how successive updates and benchmark refresh affect the evidence for improvement.

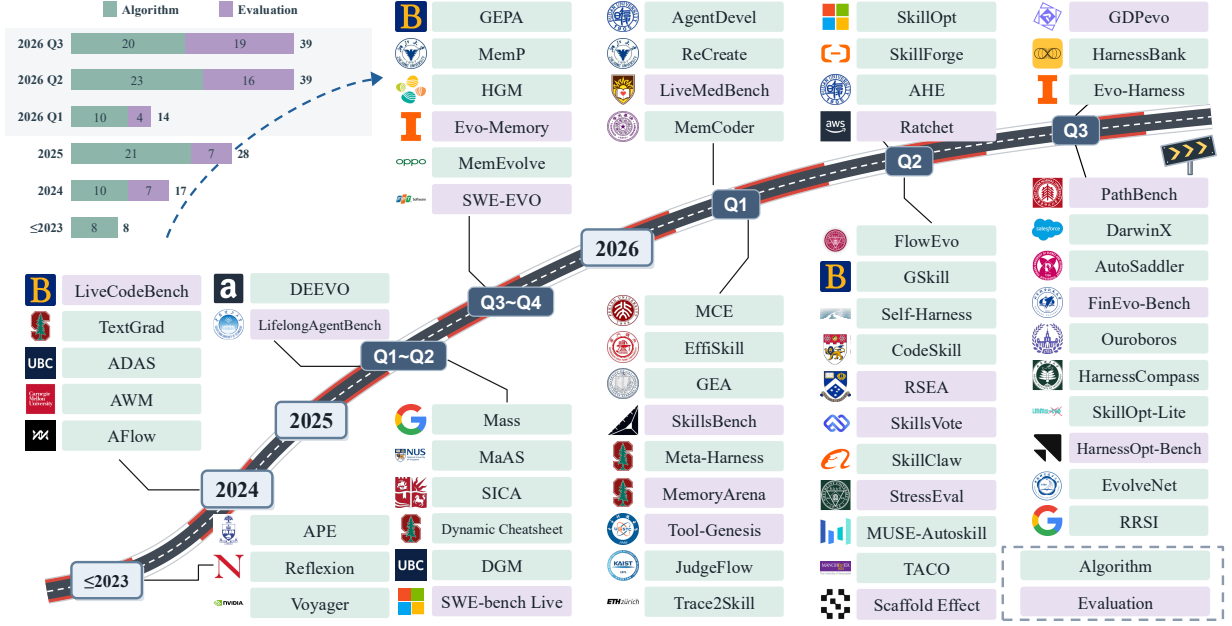


Figure 2 Growth and classification of harness self-evolution work. The inset counts each work drawn in Figure 3 once, at the period of its first public version, split by primary emphasis into update algorithms and evaluation, for works released through September 2026; every date comes from a public record except the one product page dated from the authors’ information, and the ledger in the repository names it. The road places representative systems along the same periods, tracing the field’s progression from early prompt, memory, and workflow systems to executable components and runtime logic; the systems it names are examples rather than the roster the inset counts. The classification is descriptive, and inclusion does not imply continuing self-evolution or measured improvement.

This article makes three contributions:

- **Scope and Operational Taxonomy (§2–§4):** We define harness self-evolution as a persistent operational feedback loop and hold the base model fixed when evaluating harness changes. We then organize the design space by update target, separating revisions to guidance and coordination specifications interpreted by fixed components from revisions to executable implementations, and review offline antecedents alongside the methods that reuse their mechanisms.
- **Synthesis and Empirical Review (§3–§4, §6–§7):** Using this taxonomy, we compare existing methods and find that mutation scope and persistence vary independently. We examine what their primary comparisons establish under the four evidence criteria and state the controls needed when the harness and the benchmark change together. Across deployment settings, admission checks must reflect the cost of a wrong update.
- **Theoretical Perspectives (§5):** We map harness search onto evolutionary paradigms and analyze generalization and retention from a continual PAC-learning perspective. The analysis gives a sufficient condition under which an admission comparison certifies improvement on a round’s task mixture, and shows that repeated executions of one task reduce execution noise but add no independent tasks.

Organization. Section 2 formalizes the operational lifecycle of harness self-evolution. Sections 3–4 classify interpreted and executable update targets. Section 5 develops the evolutionary and continual PAC-style theoretical perspectives. Section 6 examines empirical evaluation practices and dynamic benchmarking. Sections 7–9 cover deployment regimes, societal implications, and open research frontiers, and Section 10 concludes.

Table 1 How recent surveys cover the harness self-evolution lifecycle. Each cell records the strongest treatment found in the survey: ✓ a dedicated section or subsection that reviews multiple methods, ◦ a substantive discussion without a dedicated treatment (including future work), and — no body-text treatment beyond a table entry, passing clause, or explicit scope exclusion. For *base model held fixed*, ◦ denotes a definitional or scope-based separation; ✓ would require an experimental control, which none of these surveys supplies. For the four *evaluation claims*, ✓ requires an explicit evidentiary criterion rather than inference from a terminal task score. For *theory*, ✓ requires formal results developed by the survey itself; *societal impact* excludes technical safety. Columns follow preprint order. The Zhou et al. columns are the externalization review [11] and the coding-agent survey [15], whose declared scope is software engineering alone, so its ✓ for applications covers deployed products in that one domain.

Dimension	Gao [3]	Fang [4]	Zhou [11]	Ning [12]	Chen [14]	Ren [13]	Zhou [15]	Ours
Scope								
Base model held fixed	—	—	◦	—	—	◦	◦	◦
Update target								
Instructions and prompts	✓	✓	◦	—	—	✓	✓	✓
Context and memory	✓	✓	✓	✓	✓	✓	✓	✓
Workflow and control	✓	✓	◦	✓	◦	◦	✓	✓
Executable artifacts	✓	◦	—	◦	◦	✓	✓	✓
Evaluation claim								
Selection	—	—	—	◦	—	✓	◦	✓
Activation	—	—	—	—	—	—	✓	✓
Attribution	◦	—	◦	◦	✓	✓	✓	✓
Improvement	✓	◦	◦	◦	◦	✓	◦	✓
Analysis and implications								
Theory	◦	◦	—	◦	—	◦	—	✓
Applications	✓	✓	◦	✓	◦	✓	✓	✓
Societal impact	—	◦	◦	—	◦	—	—	✓

✓ dedicated ◦ partial — absent

2 Definition of Harness Self-Evolution

Table 2 The agent harness by component. *Runtime responsibility* states what the component must supply while a run is in progress; *mechanisms* lists the sub-functions through which it supplies it. A given implementation may fuse several components into one module or omit them entirely.

Component	Runtime responsibility	Mechanisms
Agent loop and execution control	Advances a run from one model call to the next and decides when it ends	Model invocation; tool execution; continuation and stopping rules; cancellation; error recovery
Tool and skill orchestration	Exposes external capability and reusable procedure to the model	Tool interfaces; registration and invocation; skill loading and invocation
Context construction	Assembles what the model sees on each call	System instructions; tool descriptions; conversation history; injected context
Memory and state management	Carries state beyond the current call or session	Reusable memory and retrieval; session persistence; resumption; forking
Context compaction	Keeps the assembled context within budget as a run grows	History summarization; retention and triggering policies; tool-output pruning
Scheduling and coordination	Distributes work beyond a single loop	Sub-agent delegation; task scheduling; background execution
Execution infrastructure	Bounds what a run may do and records what it did	Sandboxing; permissions and approval policies; storage; model adapters; logging and replay

Harness. Following DeepSeek Harness’s view of an agent as a model combined with a harness [16], we define a harness H as the software and configuration surrounding a model M that govern its execution and interaction with a task environment E . Table 2 summarizes its functional components, including the agent loop, tool orchestration, memory, skills, and context compaction. These are composable capabilities rather than mandatory components of every harness. Model adapters belong to this surrounding software, while model weights belong to M . A rollout τ , comprising the interaction trajectory and its outcome, is generated as

$$\tau \sim \text{Run}(M, H, E). \quad (1)$$

A rollout is scored by an evaluation procedure J , comprising the task scorer, any verifier used to decide the outcome, and the bookkeeping that records it, giving $y = J(\tau)$. We place J outside H by stipulation, and Sections 5 and 6 keep it outside the surface a candidate may modify, so that a change in J is a change in the measurement rather than a harness update. Internal critics and provisional verifiers may belong to the mutable harness, provided an external J judges the final outcome. A system that exposes this final scoring or verification procedure to its own proposals breaks that stipulation: the code is then part of what the update may rewrite, and its scores before and after an update no longer measure the same quantity. Section 6.2.1 treats this failure separately from the statistical failures of selection, because the remedy is structural rather than statistical.

Harness self-evolution. We define harness self-evolution as an iterative optimization process in which an agent system uses batches of execution experience to revise its own harness, with the aim of improving task performance or reducing resource use, while keeping the backbone model M fixed. Section 6.2.3 evaluates performance together with inference cost. At update round k , the current harness H_k interacts with a batch of sampled task environments to produce

$$\mathcal{B}_k = \{\tau_{k,i}\}_{i=1}^b, \quad \tau_{k,i} \sim \text{Run}(M, H_k, E_{k,i}), \quad (2)$$

where b is the batch size, $E_{k,i}$ is a sampled task environment, and each rollout includes its interaction trajectory, outcome, and available feedback. The harness is then updated as

$$H_{k+1} = \text{Update}(H_k, \mathcal{B}_k). \quad (3)$$

Here, k indexes update rounds. As in minibatch training, different rounds may use different task samples or revisit samples from a fixed training set; temporal task arrivals and changing task distributions are not required. Equation (3) abstracts diagnosis, candidate generation, evaluation, and retention. The update procedure may internally maintain historical feedback and candidate archives or collect additional evaluation rollouts. Rejected revisions leave $H_{k+1} = H_k$, and single-rollout updates are recovered by setting $b = 1$.

The index i counts rollouts within a round; the b rollouts need not come from b distinct task environments. Section 5 therefore distinguishes tasks from repeated executions and makes the feedback history and candidate archive explicit in the evolution state.

Recursion refers to the feedback between harness revision and experience generation: adopted revisions govern subsequent execution, and the resulting rollouts inform further revisions. We designate an update process as harness self-evolution when two conditions hold.

1. **Operational feedback and update continuity.** Within the operating lifecycle of the agent, feedback from its own executions is used by an algorithmic process to form or select the revision, and the update mechanism remains in place after the transition, so later executions can trigger further updates. External environments and automated evaluators may supply evidence, and humans may give feedback or control admission, but a human does not simply author the

payload and install it as the next state. This excludes manual configuration, and design-time optimization that terminates after exporting a frozen artifact: an offline optimizer removed once it has delivered a static prompt, workflow, or program leaves a harness behind but no loop, and Sections 3 and 4 mark such systems as offline antecedents.

2. **Persistence across runs.** When a candidate is accepted, the resulting harness state is persisted and actually loaded by a subsequent run, so that it can affect rollouts it did not generate. This excludes candidates evaluated and discarded inside a search, within-run reflection that never changes later execution, and passive logs that are stored but never read by the harness. Whether the loaded state then operated in those runs is the activation claim of Section 6.1, not a condition of scope.

These conditions define a repeatable capability rather than a minimum number of observed rounds. An experiment may contain a single operational update and still demonstrate the mechanism, provided the accepted state stays inside the same feedback and update lifecycle. Ordinary execution under unchanged rules, such as appending a session log or triggering a fixed compaction policy, meets neither condition, because nothing was revised; revisions may instead change those rules, or distill reusable knowledge and skills that guide later execution. Methods that revise the update mechanism itself add a further level of recursion, and Section 4.1 covers the cases that change a memory-management process rather than its contents. Improvement is the objective of the process, not a guarantee assumed for every update: sampling independence, generalization, and retention require separate analysis.

Naming. This process is called harness self-evolution in the title of this article and harness RSI, for recursive self-improvement, in much of the literature. We retain harness RSI as a conventional short name for this research area, not as a claim that every accepted revision improves performance. We use *self-evolution* for the update mechanism whether or not the accepted state performs better, and reserve *self-improvement* for a beneficial change supported by an explicit comparison.

For systems that also train the model, we review the harness-side update under the two conditions above and assess comparisons that hold the task model fixed. Weight-only updates and fixed components without an update mechanism are outside this scope.

Sections 3 and 4 also review the offline antecedents of these mechanisms—prompt, workflow, and program search that export a frozen artifact—together with boundary cases such as tool synthesis whose output never persists. The prose identifies these lifecycle differences when introducing each work.

Figure 3 maps the literature reviewed under this definition onto three of the sections that follow. Its branches separate what an update may change—interpreted payloads in Section 3 and executable implementations in Section 4—from what an evaluation may claim in Section 6. How the search over candidates is organized is taken up separately in Section 5. The components of Table 2 are the hosts of that first split: Section 3 covers the payloads they interpret, and Section 4 covers the case where a component is itself what the update rewrites. Applications and governance lie outside this map and are treated in Sections 7 to 9.

3 Interpreted Update Targets

Interpreted updates revise instructions, memory, and workflow specifications while leaving the model, native tools, and host implementations fixed. Their effects depend on how those fixed components read the revised content or enact the revised coordination. Figure 4 places these targets alongside the executable changes of Section 4.

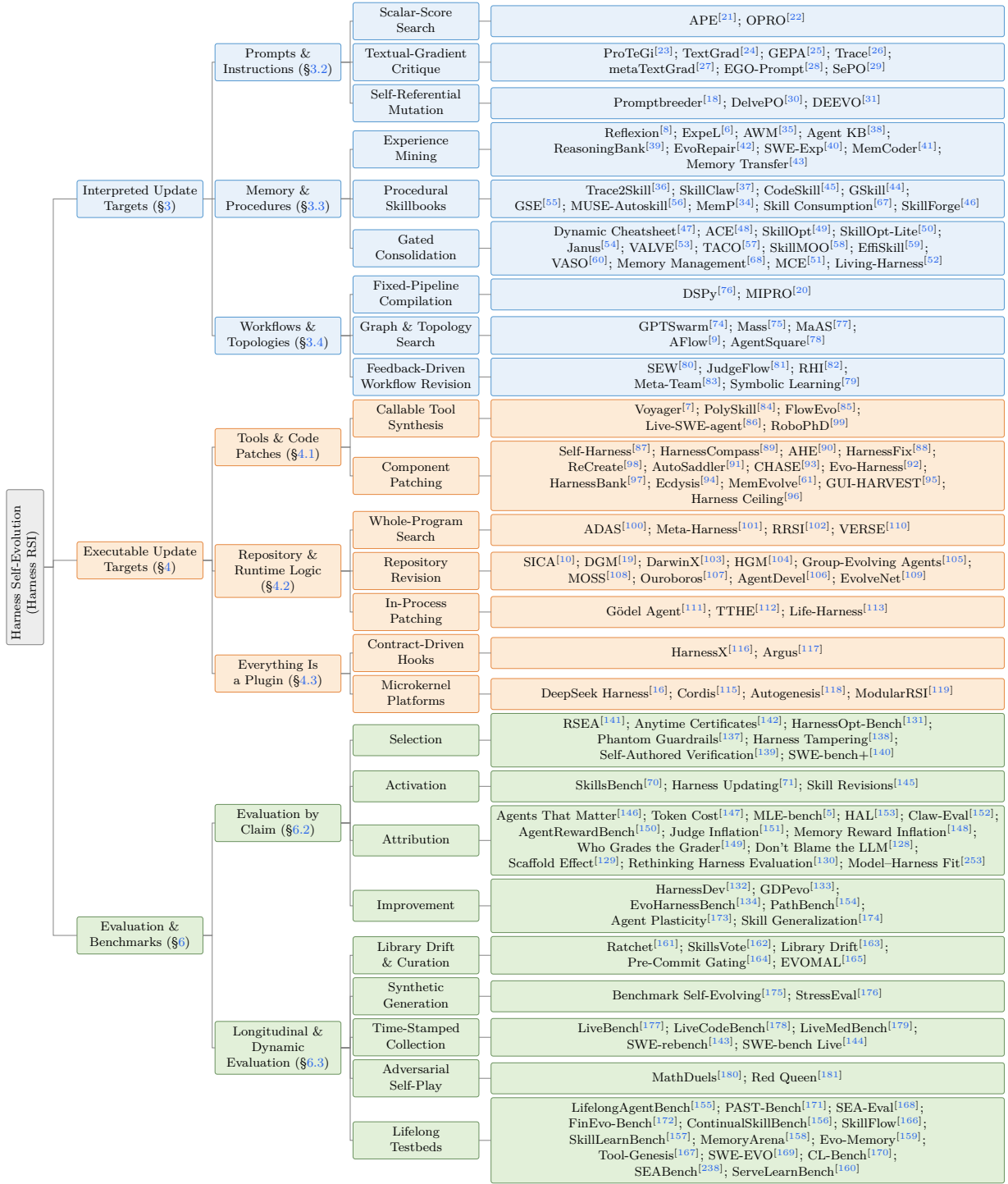


Figure 3 Taxonomy and citation map of harness self-evolution (§3, §4 and §6). Branches correspond to sections, second-level nodes summarize subsections, and third-level nodes identify update mechanisms, evaluation claims, or evaluation settings. A work is listed once, under what it contributes, so the systems §6 uses as examples appear under the update target they change rather than under evaluation. Deployed products are cited in §7 instead, because they persist state across sessions without reporting the feedback-driven update loop of §2; DeepSeek Harness is the one product listed here, beside the Cordis kernel paper, as the plugin platform of §4.3.

3.1 The Rise of Harness RSI

The idea of a machine improving its own problem-solving machinery predates language models. Good’s 1965 argument placed machine design among the intellectual activities an advanced machine could perform: a better designer could construct a better successor, which could in turn improve the design process [1]. This was a prospect of recursive amplification rather than an implemented update procedure. EURISKO later made heuristic discovery an operational part of search, learning domain concepts and heuristics, including heuristics for modifying other heuristics [17]. Its relevance is the change in what learning may act on: the procedures that guide future search can themselves become objects of search.

The Gödel Machine addressed self-revision through a different mechanism. Its initial program contains a proof searcher and an axiomatic description of the hardware, software, and utility; it may rewrite any part of that software, including the proof searcher, once the usefulness of doing so is proved [2]. The formal guarantee is relative to the encoded assumptions and what can be proved within them. This connects the benefit of a rewrite to the cost of continuing to search for alternatives. EURISKO and the Gödel Machine thus offer distinct precedents: empirical revision of search heuristics and formally justified revision of the problem solver itself. Neither requires the modern separation between a pretrained model and its surrounding harness.

Language models make that separation a practical site of adaptation. A fixed model can interpret revised instructions, summarize execution experience, and generate callable code, allowing feedback to change later behavior through external artifacts. Reflexion stores textual feedback for another attempt, ExpeL extracts reusable insights across tasks, and Voyager retains executable skills [6–8]. These systems expose different routes from experience to future execution, with within-task reflection supplying a precursor to the persistent lifecycle defined in Section 2. Promptbreeder extends the mutable state to the prompts that generate further mutations [18]; repository-level systems such as the Darwin Gödel Machine extend it to the agent’s implementation, using empirical evaluation and an archive of alternatives [19]. The locus of revision can therefore move from guidance for a fixed executor to the machinery that implements execution and subsequent updates.

Harness RSI brings these developments together around a concrete question: what learned state does the next run inherit? Keeping the task model fixed makes instructions, memories, workflows, and executable components available for separate inspection and revision. The historical continuity lies in making problem-solving procedures revisable; the contemporary form uses model-generated proposals and execution feedback to maintain those procedures as external state. The following subsections and Section 4 distinguish the resulting update targets and their lifecycles.

3.2 Instructions and Prompts

The first target is text supplied directly to the base model: system instructions, prompt templates, few-shot demonstrations, and mutation specifications. Updating these artifacts changes the context that steers token generation, not the native tools, scheduling policy, or host runtime [20]. The effect is therefore behavioral rather than enforceable: an instruction can shift output tendencies, but it cannot by itself impose a runtime invariant, rigid schema, or execution boundary.

Score-guided search. Black-box prompt search evaluates candidate instructions through task scores. APE samples instructions from input–output examples, while OPRO conditions its optimizer on a history of scored candidates [21, 22]. The score identifies promising candidates, but supplies little information about which instruction caused a failure. Search must infer a useful revision from the candidate text, examples, and evaluation history available to it.

Trace-guided revision. Execution traces expose intermediate decisions and failures that a final score leaves unexplained. ProTeGi uses critiques to guide prompt edits, TextGrad propagates

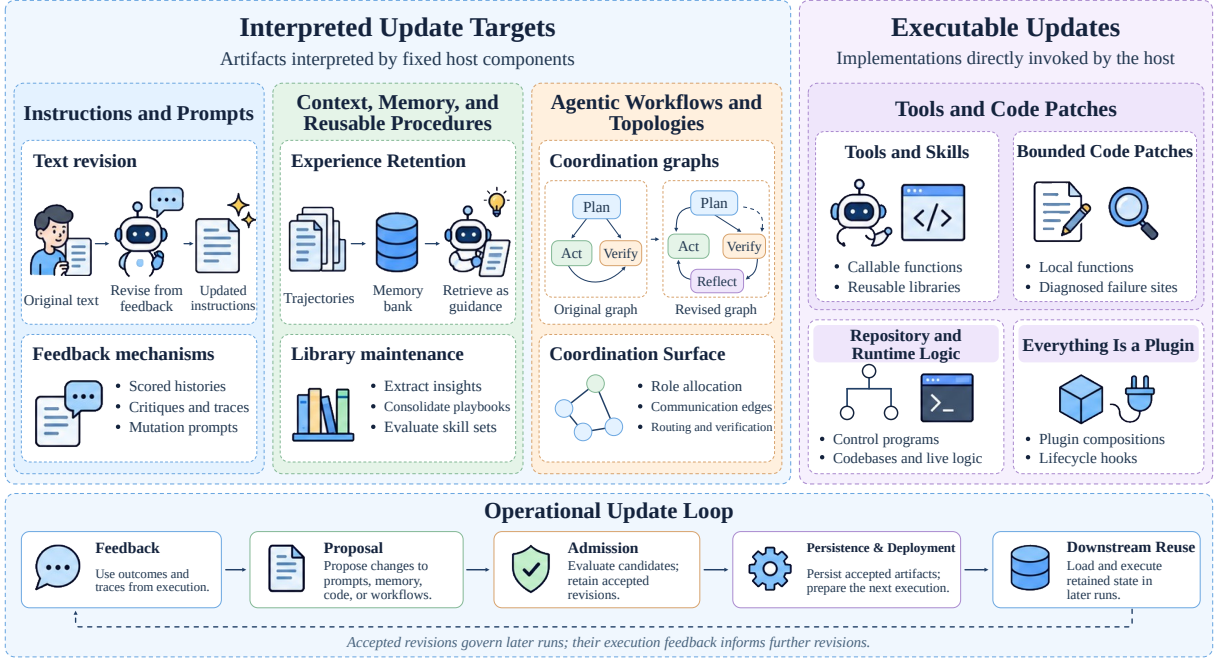


Figure 4 Harness update targets and the operational update loop. Under a fixed base model, interpreted targets comprise instructions and prompts, persistent memory and reusable procedures, and workflow specifications enacted by fixed host components. Executable targets cover tools and code patches, repository and runtime logic, and plugin implementations and compositions. The first two describe mutation scope; plugin interfaces organize binding and replacement across these scopes. Callable tools and bounded patches share a fixed surrounding executor. The lower panel connects execution feedback, proposal, admission, persistence and deployment, and downstream reuse. Accepted revisions govern later executions, whose rollouts inform further revisions. Persistence and deployment alone establish neither operational activation nor improvement.

textual feedback through a computation graph, and GEPA uses execution feedback to evolve complementary prompts [23–25]. Trace likewise organizes optimization around graph-structured computations [26]. These methods give the proposer evidence about where a procedure failed, so it can revise the instruction associated with that failure. Related methods adapt optimizer prompts, domain-specific prompts, and long-lived system instructions [27–29].

Updating the proposer. Promptbreeder co-evolves task prompts and the mutation prompts used to revise them; later variants guide mutation directions or compare candidates through tournaments [18, 30, 31]. Retaining a useful mutation prompt changes how later candidates are generated. This adds adaptation to the search procedure, while the resulting task instructions remain text interpreted by the fixed model. Most prompt optimizers freeze those instructions after search; operational self-evolution also retains the mechanism that revises them from later experience.

Carrying prompt revision into continued operation creates a context-management problem. Accumulated instructions consume attention and inference resources and can reduce retrieval precision as context grows [32, 33]. Curation and external memory address this accumulation; the following subsection examines how systems maintain guidance beyond a single prompt.

3.3 Context, Memory, and Reusable Procedures

Persistent memory stores experience-derived guidance outside the immediate context window and retrieves it into later runs. Its contents range from episodic reflections and cross-task strategies

to playbooks and reusable skill files [11, 34]. The learning problem includes both forming these entries and deciding which ones to retain and retrieve.

Forming reusable guidance. The intended reuse horizon distinguishes early memory systems. Reflexion stores feedback for another attempt at the same task [8]. ExpeL and Agent Workflow Memory extract insights or recurring sub-goal procedures for reuse across tasks [6, 35]. Trace2Skill and SkillClaw package experience as modular skill documents [36, 37], making the reusable unit a procedure that can be maintained within a larger library.

Reusable guidance can span domains or remain tied to one environment. Agent KB and ReasoningBank retain cross-domain experience or reasoning strategies [38, 39]; software-repair memories retain repair outcomes and repository structure [40, 41]. EvoRepair scores and reuses vulnerability-repair experience across long-horizon repairs under a fixed executor [42]. Work on behavioral-rule transfer tests whether extracted guidance remains useful outside its source setting [43]. Repository-specific skills make a different choice: gskill learns Markdown documents containing a codebase’s conventions, testing patterns, and modification procedures from executions on generated repository tasks [44]. The reusable knowledge is tied to the codebase even when a different agent reads it.

Turning an episode into a procedure requires choosing which details to preserve and when the resulting guidance applies. Memp compares storing successful trajectories, abstracting them into natural-language scripts, and pairing the scripts with the original trajectories; its update policies include filtering new entries by successful execution and revising a retrieved memory using the error trace from a failed attempt [34]. CODESKILL separates task-level strategies from event-driven instructions for recurring command errors or test-output patterns. With the downstream coding agent frozen, its trained manager revises a skill’s trigger conditions or instructions, then adds the candidate, merges complementary guidance, or rejects redundant and weakly grounded candidates [45]. SkillForge instead mines historical support tickets into branching handling procedures with tool guidance and escalation fallbacks. Later failures are aggregated and mapped to missing, insufficient, or incorrect passages in the skill; its optimizer makes targeted edits under an instruction to preserve existing working content [46]. These approaches retain different units of procedural knowledge: an example with an abstraction, a condition-linked instruction, or a domain workflow that can be repaired at a specific branch.

Maintaining the library. As experience accumulates, the updater must decide what to add, revise, combine, or remove. Dynamic Cheatsheet and Agentic Context Engineering (ACE) rewrite and consolidate accumulated strategies, while SkillOpt, SkillOpt-Lite, and Meta Context Engineering (MCE) search revised skill sets or archived candidates before replacing the active library [47–52]. Admission can be separated from the rule that generates these revisions. VALVE uses a statistical gate over paired outcomes on a fixed validation panel [53]; Janus wraps an existing updater and, when a candidate deviates from the recent update direction, compares the previous and candidate memories on coverage, boundary, and fresh tasks before deploying either [54].

Maintenance can also change how procedures are organized. GSE consolidates trace-local proposals and updates a graph of dependencies, co-usage, and conflicts alongside skill contents, so revising one procedure can trigger changes to related ones [55]. MUSE-Autoskill keeps usage notes with each skill and supports refinement, merging, and pruning from the active catalog [56]. Its skill packages may also contain executable scripts; the instructions and notes are interpreted state, while revisions to those scripts cross into the executable targets of Section 4.1.

Library optimization can also target compression or competing objectives [57–59], or constrain revisions through specifications [60]. Changing the machinery that maintains the library crosses a different boundary: MemEvolve’s outer loop rewrites the code that encodes, stores, retrieves, and manages memories, which we classify with the executable targets of Section 4.1 [61].

Use in later runs. Project memories and file-based skills also persist across sessions in deployed coding agents [62–65]. Engineers may maintain such rules manually by codifying accepted review comments [66]; this is configuration under Section 2, rather than algorithmic self-evolution. Measurement studies examine how generated skills are used and which management choices affect outcomes [67, 68]. Whether authored by people or generated by a loop, the stored guidance is interpreted probabilistically [69]: a useful skill may go unretrieved or be ignored after retrieval [70, 71]. Stale strategies and prompt-injection payloads can persist through the same mechanism [72, 73], making conflict detection and regression checks part of library maintenance [54].

3.4 Agentic Workflows and Topologies

Workflow and topology targets specify how fixed model and tool components coordinate across multiple steps: role allocation, communication graphs, conditional routing, and verification or termination cycles [74, 75]. We restrict this category to coordination specifications enacted by a fixed orchestration framework. Whether represented as text, graphs, or code, the mutable specification determines task decomposition, communication, routing, and stopping. Node implementations, tool interfaces, and the orchestration framework remain fixed.

Searching coordination. Workflow methods expose different coordination choices to search. DSPy optimizes prompts and demonstrations within a programmer-defined pipeline; MIPRO jointly searches these choices across pipeline stages [20, 76]. GPTSwarm learns connections between fixed operations, Mass searches prompts and communication topologies, and MaAS samples query-dependent architectures from predefined operators [74, 75, 77]. AFlow searches code that composes reusable operators [9]. Its code changes their arrangement and prompts while leaving operator implementations fixed, which places it on the interpreted side of the taxonomy.

AgentSquare and Symbolic Learning extend workflow search into module or tool implementation: AgentSquare generates new modules as it searches compositions, and Symbolic Learning jointly revises structure, prompts, and tools [78, 79]. Their primary placement here follows their contribution to workflow optimization; their generated implementations cross into the executable surface of Section 4.1.

Feedback and persistence. The feedback horizon determines whether a searched workflow becomes continuing operational state. SEW mutates prompts and topology across alternative representations, and JudgeFlow uses outcomes and block-level diagnoses to revise sequences, loops, and conditionals [80, 81]. RHI revises roles, instructions, communication contracts, and interaction hops within one task episode [82]. Meta-Team carries experience across tasks in retained prompts, teammate profiles, collaboration notes, and coordination rules [83].

Most workflow methods export a pipeline or revise it within one optimization episode. In Meta-Team, later tasks inherit coordination state and contribute experience to its next revision. This continuity depends on what the system retains and updates after search.

Instructions steer individual model calls, memory preserves experience for later retrieval, and workflow specifications organize coordination among fixed operations. Across these targets, the inherited state changes what the existing executor is asked to do. Section 4 turns to changes in the implementations that supply those operations and enact their coordination.

4 Executable Update Targets

Executable updates change tools, middleware, control programs, repository code, or runtime component bindings [12]. We organize them by component implementations, repository and runtime logic, and plugin implementations and compositions. The first two describe the scope of revision: a bounded component or the control program that coordinates several harness

functions. The third describes how implementations are exposed, bound, and replaced through declared interfaces, and can support either scope. A work is placed by its main contribution, with overlapping mutable surfaces described where they matter to its operation.

4.1 Tools and Code Patches

A callable tool, function, or designated component file gives the update loop a bounded executable target. Keeping the surrounding executor fixed makes testing and rollback local to that unit, although faults can still propagate through its outputs or shared state.

Tool synthesis. Callable-tool systems turn successful execution into independently invoked code. Synthesized functions may be retained for later retrieval or composition while the scheduler and tool-loading machinery remain fixed [7, 84]. FlowEvo makes the boundary with Section 3.4 explicit: it compiles a successful workflow into a validated callable artifact, stores it in a persistent skill bank, and later either executes it directly or supplies its structured description to workflow generation [85]. The first route is an executable update; the second remains an interpreted use of the same record. Live-SWE-agent provides the contrasting lifecycle: its generated Bash and Python tools are discarded with the task container, so it demonstrates tool synthesis without persistent harness evolution [86].

Component revision. Bounded patching systems diagnose failures and revise declared components. Agentic Harness Engineering (AHE) uses runtime observability to identify local code or configuration changes; related methods constrain edits through placement rules or workspaces covering tools, middleware, memory, prompts, and subagent configuration [87–94]. GUI-HARVEST uses repeated executions of a task to identify recurring failure modes and records an edit’s predicted behavioral effect before testing it [95].

MemEvolve exposes a memory provider’s encode, store, retrieve, and manage implementations to its outer update loop [61]. The mutable object is thus the code that manages experience, rather than the memory entries reviewed in Section 3.3. Bounded edits can also probe the limits of harness adaptation: a long-horizon planning study uses them to distinguish process failures repaired by the harness from failures addressed by weight training [96].

Admission and composition. Admission and bookkeeping vary within this common mutation boundary, from regression checks and separate validation tasks to file-level rollback. HarnessBank maintains an explicit record: candidates are held in an archive partitioned by semantic descriptor, each must declare the trace beacon its patched path emits, and the evaluation code itself sits in a kernel candidates cannot modify [97]. Sections 6.2.1 and 6.2.2 take up the kernel and the beacon as evidence requirements. CHASE changes what the gate asks: a Challenger proposes benchmark-protocol transformations intended to destroy the candidate’s gain over the parent rather than its raw score, and a patch is admitted only if its gain survives every transformation confirmed into the archive [93].

Some systems revise interpreted and executable artifacts together. AgentSquare and Symbolic Learning generate modules or tools alongside the workflow changes described in Section 3.4 [78, 79]. ReCreate edits roles, processes, tools, and memory at textual and code levels, while keeping execution-environment and system-infrastructure adaptation outside its scope [98]. RoboPhD co-evolves a deterministic Python database-analysis tool and model-readable SQL instructions; its deployed package no longer depends on the evolutionary infrastructure [99]. These systems show how a bounded tool implementation can complement guidance interpreted by the model.

4.2 Repository and Runtime Logic

This category makes the end-to-end control program mutable. Unlike a local patch whose enclosing interfaces remain fixed, an update may cross component boundaries and revise context construction, tool dispatch, verification and recovery, the agent loop, or the code that proposes later changes. The update boundary spans several harness functions, whether the substrate is a single program, a multi-file repository, or live process memory.

Whole-program search. One mechanism synthesizes a complete control program inside a fixed outer search loop. ADAS uses a fixed meta-agent to generate agent **forward** implementations and retains evaluated candidates in an archive [100]. Meta-Harness gives an external coding agent the source, scores, and traces of prior candidates and asks it to synthesize a complete task-specific harness program [101]. RRSI regularizes the same search with annealed edit budgets, leakage screening, and noise-adjusted acceptance floors, trading gain on the evolution set for out-of-distribution transfer [102]. All three are offline antecedents in the sense of Section 2: their output is an executable harness selected for later deployment, while the deployed harness does not continue updating from new task experience.

Repository revision. Versioned-repository methods retain implementations from which later revisions can branch. SICA exposes its multi-file Python codebase, including tools, subagents, and control flow, to the best archived agent [10]. The Darwin Gödel Machine (DGM), DarwinX, and the Huxley-Gödel Machine (HGM) preserve alternative lineages for selection and inheritance [19, 103, 104]; Group-Evolving Agents lets selected parents share patches, tools, and execution experience before producing framework-level revisions [105]. The archive supports future search, while promotion determines which candidate actually becomes the runtime.

Promotion mechanisms include release-candidate regression, reviewed commits, health-checked container replacement, and validation before a merged program is broadcast as shared state [106–109]. MOSS rewrites routing, hooks, state management, and dispatch code, then requires a health check and human authorization before replacing the running container. VERSE also makes the optimizer mutable: it revises the executor’s harness and its own prompts, tools, hooks, and notes, with execution checks on draft edits and both models’ weights fixed [110].

Runtime revision and persistence. Runtime updates can also act directly on the running process. Gödel Agent uses monkey patching to replace decision methods, action spaces, and modification routines in live Python memory; the revised methods govern later decision steps but disappear when the process ends [111]. TTHE instead evolves executable harness programs from execution traces and carries the selected program into subsequent inputs, so it is persistent across test-time rounds rather than merely an ephemeral interface change [112].

Life-Harness combines environment contracts and model-readable skills with executable action realization and trajectory regulation, then freezes the runtime interface for evaluation [113].

Runtime adaptation in HarnessOpt. HarnessOpt extends SkillOpt-Lite from editing a skill document to revising allowlisted harness scripts, with candidates checked in an isolated subprocess [50]. Its spreadsheet-agent analysis identifies different bottlenecks across models. Incomplete observations and weak final checks prompted wider spreadsheet previews and a dedicated output-inspection step. For models caught in repetitive reasoning after a tool failure, an interception policy retries with a lower reasoning-effort setting. The revisions change what information the model receives, how it checks completion, and how execution continues after failure. They complement a retained skill by changing the conditions under which its procedure is carried out.

4.3 Everything Is a Plugin

Plugin-based harnesses place executable components behind declared interfaces and lifecycle hooks [12, 114]. An update may change a plugin’s implementation or the composition that binds tools, memory, providers, agent loops, and verifiers into the runtime. These bindings select which implementation supplies a service, such as memory retrieval or tool dispatch, rather than only arranging fixed operators in a task-level graph. A fixed kernel governs the available interfaces, dependency checks, and mounting and replacement operations.

Composition and admission. DeepSeek Harness’s *Everything Is a Plugin* (EIP) architecture gives this design a concrete form. Agent capabilities are plugins, while the Cordis kernel manages their mounting, unmounting, and dependencies; append-only records support checkpointing and replay [16, 115]. Feedback-driven systems use plugin interfaces to revise implementations or compositions under typed hooks, dependency constraints, and activation checks [116–119]. HarnessX, for example, checks hook types, singleton constraints, and smoke tests before committing an AEGIS-proposed configuration. These systems add proposal and admission policies to the substrate’s operations for binding and replacing components.

Runtime composition in DSH Everything. DSH Everything makes the binding between a capability and its executable implementation an explicit update target [120]. In its Codex reference patches, a common capability view supplies both the tool declarations shown to the model and the routes used to execute its calls. Each model request captures a stable snapshot, so calls produced by that response use the versions declared when the request was made. Catalog replacement checks the expected version before committing a new set of bindings, which becomes available to the next step. This lets the available capabilities change during a session while the native agent loop remains fixed. The example implements the catalog and binding mechanisms; the surrounding protocol assigns candidate generation, isolated validation, worker activation, evaluation, and promotion to an external runtime.

Validation and recovery. Declared interfaces make compatibility and replacement easier to check. Behavioral interactions still arise through outputs and shared state, so a candidate must be assessed in its intended composition. Checkpoints and dependency reconciliation support recovery within runtime-managed state [115]; external actions require pre-execution checks or, after an effect has occurred, compensation. These operational responsibilities connect the plugin boundary to the reuse and deployment choices in Section 9.

The update target is the machinery used to perform tasks. Harness-of-Harness instead revises the task’s software deliverable while keeping its harness–model configuration fixed [121]. Mutation scope and persistence vary independently within these targets. Repository and runtime logic alone illustrates the range: ADAS, Meta-Harness, and RRSI select programs offline, MOSS promotes revisions into a running container, and Gödel Agent’s live patches disappear with the process. A comparison therefore needs to identify both the editable surface and the lifecycle through which an accepted change reaches later execution.

5 Theory of Harness Self-Evolution

We examine harness RSI from two theoretical perspectives: optimization and generalization. Turing-complete harness languages can express arbitrary computable control logic in principle, although interfaces, permissions, and resource budgets constrain deployed systems. Expressivity alone does not explain how to find a useful program or establish that it helps. Many reviewed methods perform derivative-free search over discrete harness state, using objective evaluations and rollout feedback to guide revisions (Section 5.1). Continual harness evolution then raises questions

of generalization and knowledge retention, for which PAC learning and its continual-learning formulations provide useful theoretical foundations (Section 5.2). Recent work makes these limits explicit: Harness Evolution as Learning studies approximation, generalization, and optimization constraints, while Reliable Self-Evolution derives adoption conditions whose ceiling depends on verification and evaluation cost [122, 123].

5.1 An Evolutionary Perspective on Harness Search

Harness self-evolution can be understood through the evolutionary algorithm (EA) framework of representation, variation, evaluation, and selection. A candidate encodes an executable harness configuration, which may include instructions, tool interfaces, control flow, or reusable skills. Execution on a collection of tasks supplies empirical fitness, while an optimizer proposes modifications and decides which candidates to retain. Empirical fitness supports selection, while traces, diagnostics, and natural-language critiques can guide proposal generation. These methods generally do not differentiate task performance through the discrete prompt, workflow, or code edit. The EA framework captures shared search structures without reducing the available feedback to a scalar or asserting that every harness update implements an evolutionary algorithm. For skill evolution, the search can operate on an individual skill or a skill library within the surrounding harness. Table 3 organizes these methods by the evolutionary paradigms and mechanisms they instantiate or resemble. We use the broader EA framework because the surveyed methods include evolution strategies, archive-based program search, and quality-diversity search, beyond conventional genetic algorithms. The categories are not mutually exclusive: a system may combine one selection strategy with several variation and resource-allocation mechanisms.

The simplest correspondence is elitist single-offspring search: a candidate is generated from the incumbent and replaces it when evaluation indicates an improvement. MCE explicitly adopts a history-informed $(1 + 1)$ evolution strategy [51], while SICA, SkillOpt, SkillOpt-Lite, and the skill-refinement path of SkillClaw exhibit related selection structures [10, 37, 49, 50]. These methods differ in how they generate edits and use feedback, but share the principle of retaining a preferred incumbent across iterations. Self-Harness extends this pattern by proposing multiple edits from a common incumbent and evaluating them before promotion [87]. Its search therefore resembles an elitist $(1 + \lambda)$ scheme, although merging several compatible accepted edits differs from selecting a single survivor in canonical $(1 + \lambda)$ -ES.

Archive-based methods broaden the search beyond the current incumbent. DGM, DarwinX, and HGM retain candidate programs and organize subsequent exploration through parent selection, modification, and lineage expansion [19, 103, 104]. Their correspondence is to evolutionary search over programs; this does not imply the use of classical tree-based genetic programming operators. HarnessBank further structures its archive using semantic descriptors and competition within cells, following a MAP-Elites-style quality-diversity design [97]. This organization preserves candidates addressing different mechanisms instead of ranking the entire archive solely by one global score. HGM adds a complementary mechanism: Thompson sampling allocates lineage-expansion and candidate-evaluation budgets, coupling evolutionary search with bandit-based resource allocation [104].

Variation also differs from conventional fixed mutation and crossover operators. In these systems, language models can use execution traces and failure diagnoses to propose targeted changes to natural-language instructions or executable code. Recombination combines content from different candidates, as in DarwinX, MCE, and HarnessCompass, or merges compatible edits, as in Self-Harness [51, 87, 89, 103]. The strength of this correspondence depends on the surrounding search procedure. For example, Trace2Skill consolidates trajectory-derived patches, which resembles recombination at the level of information reuse, but patch consolidation alone does not establish a complete evolutionary selection loop [36]. Similarly, HarnessCompass’s component-wise optimization and subsequent combination resemble cooperative coevolution, while

lacking an explicit implementation of independently evolving subpopulations [89].

Selection need not reduce candidate quality to a single task score. Meta-Harness considers performance–cost trade-offs, while Self-Harness requires non-degradation on both held-in and held-out splits and improvement on at least one [87, 101]. SkillMOO applies the same principle one level down, selecting among skill sets under several competing objectives rather than one [58]. Meta-Harness and SkillMOO consider competing objectives, while Self-Harness applies a dominance-like acceptance condition across evaluation splits. These are correspondences at the selection level, rather than evidence that the systems implement a complete algorithm such as NSGA-II. Together, the mappings in Table 3 distinguish how systems generate variation, preserve alternatives, and allocate evaluation effort. They also expose a shared limitation: evolutionary selection operates on observed fitness, so retaining an empirically better candidate does not by itself establish improvement on unseen tasks or preservation of earlier capabilities. The following subsection examines these generalization and retention requirements.

5.2 Generalization and Retention from a Continual PAC-Learning Perspective

Evolutionary search explains how harnesses achieve higher empirical fitness, but empirical selection alone does not establish improvement on unseen tasks. Successive revisions must also preserve previously acquired capabilities. The continual PAC-learning framework motivates studying these requirements jointly [124]. We use this perspective to distinguish sample stability, historical retention, and reliable candidate selection. The bounds below give sufficient conditions under explicit sampling assumptions; they do not establish PAC learnability of the surveyed systems.

Continual refinement and its scope. Let X_k contain the complete evolution state after round k , including the current harness, skill library, candidate archive, and feedback memory, and let $H_k = \text{Deploy}(X_k)$ denote the resulting executable system. Write $X_k = \mathcal{A}_k(X_{k-1}, D_k; U_k)$, where D_k contains n_k independent tasks from $\mathcal{D}_k$ and U_k represents auxiliary randomness and any external evaluation samples. For the stability argument, task draws are independent within and across rounds, the distributions are exogenous, and auxiliary random sources are independent of these draws. Feedback generated from a task remains part of the evolution process, rather than an independent observation. We hold the backbone model and evaluator fixed and use a bounded task loss $\ell(H, z) \in [0, 1]$, averaged over execution randomness. Define

$$R_s(H) = \mathbb{E}_{z \sim \mathcal{D}_s} \ell(H, z), \quad \hat{R}_s(H) = \frac{1}{n_s} \sum_{i=1}^{n_s} \ell(H, z_{s,i}). \quad (4)$$

This formulation covers both skill updates and broader harness changes. Adaptive curricula, such as Voyager’s [7], require an extension of the exogenous-sampling model, or a separate evaluation stream.

Stability and reusable structure. Algorithmic stability relates sensitivity to individual observations to generalization [125]. Let $H_k^{(s,i)}$ be obtained by replacing $z_{s,i}$ with an independent task from $\mathcal{D}_s$ and rerunning the entire downstream evolution process with coupled random sources. In particular, later feedback, archive contents, and selection decisions must be recomputed. Define

$$\beta_{s,k} = \frac{1}{n_s} \sum_{i=1}^{n_s} \mathbb{E} \left[\left| \ell(H_k, z_{s,i}) - \ell(H_k^{(s,i)}, z_{s,i}) \right| \right], \quad s \leq k. \quad (5)$$

A replace-one exchangeability argument then gives

$$\left| \mathbb{E} \left[R_s(H_k) - \hat{R}_s(H_k) \right] \right| \leq \beta_{s,k}. \quad (6)$$

Table 3 Harness and skill optimization methods mapped onto evolutionary algorithm paradigms. *Mapping rationale and scope* states what each correspondence rests on and where it stops: a method appears in a row when its search exhibits that paradigm’s structure, which is weaker than implementing the algorithm. Rows are not mutually exclusive, so a method may appear in several. Section 5.2 takes up the limitation they share, that selection acts on observed fitness.

EA Paradigm / Mechanism	Mapping Rationale and Scope	Representative Methods
Elitist $(1 + 1)$ search / evolution strategies	A single candidate is compared with the incumbent, retaining the better solution. MCE explicitly adopts a history-informed $(1 + 1)$ -ES; the other methods share its selection structure. For SkillClaw, this mapping applies to refinement of existing skills.	SICA [10]; MCE [51]; SkillOpt [49]; SkillOpt-Lite [50]; SkillClaw [37]
Elitist $(1 + \lambda)$-style search	Multiple edits are proposed from the same incumbent and evaluated before acceptance. Merging compatible accepted edits differs from canonical $(1 + \lambda)$ -ES survivor selection.	Self-Harness [87]
Archive-based evolutionary program search	Programs serve as evolving individuals. Parent selection, code modification, and lineage retention organize the search. This is a program-level EA correspondence, rather than evidence of classic tree-based GP operators.	DGM [19]; DarwinX [103]; HGM [104]
Recombination / crossover	Complementary content from prior candidates is combined. MCE explicitly defines agentic crossover. Trace2Skill provides only a patch-consolidation analogy, without a complete evolutionary selection loop.	DarwinX [103]; HarnessCompass [89]; Self-Harness [87]; MCE [51]; Trace2Skill [36]
Quality-diversity / MAP-Elites	Semantic descriptors partition the archive into cells, with elites retained within each cell. HarnessBank explicitly draws on MAP-Elites, while modifying parent selection and adding verification gates.	HarnessBank [97]
Cooperative coevolution	Component-wise optimization followed by combination resembles cooperative coevolution. The correspondence is limited: independent evolving subpopulations are not explicitly maintained.	HarnessCompass [89]
Pareto selection in multi-objective EAs	Performance–cost frontiers or multi-criterion acceptance gates resemble Pareto-based selection. This is a selection-level correspondence, not an implementation of a complete multi-objective EA such as NSGA-II.	Meta-Harness [101]; Self-Harness [87]; SkillMOO [58]
Bandit-guided evolution	Thompson sampling allocates lineage-expansion and candidate-evaluation budgets, combining evolutionary search with bandit-based resource allocation.	HGM [104]

This controls the absolute value of the expected signed gap, not the expected absolute gap or a high-probability deviation. Trace2Skill’s aggregation of trajectory-local lessons [36], SkillOpt’s structured refinement [49], and PolySkill’s polymorphic abstraction [84] are consistent with seeking reusable structure. However, abstraction and small edits do not themselves bound $\beta_{s,k}$: a single observation can still change a shared rule or the selected lineage. Establishing stability requires algorithm-specific analysis; task-replacement experiments provide an empirical sensitivity diagnostic, rather than a proof.

Retention requires controlling accumulated regressions. Generalization and retention are distinct. Define $F_{s,k} = R_s(H_k) - R_s(H_s)$, where positive values indicate forgetting and negative values indicate backward transfer. Equation (6) implies the decomposition

$$\mathbb{E}R_s(H_k) \leq \mathbb{E}\hat{R}_s(H_s) + \beta_{s,s} + \mathbb{E}F_{s,k}. \quad (7)$$

This is not a retention guarantee unless forgetting is separately controlled. For example, if each subsequent update satisfies $R_s(H_u) - R_s(H_{u-1}) \leq \varepsilon_{s,u}$, then telescoping gives

$$F_{s,k} \leq \sum_{u=s+1}^k \varepsilon_{s,u}. \quad (8)$$

Thus, even small tolerated regressions can accumulate. Self-Harness uses split-wise non-degradation checks [87], HarnessX tests previously passing tasks and isolates variants [116], and AHE uses execution evidence to identify and roll back harmful changes [90]. These mechanisms address empirical retention, but historical replay alone does not certify the population inequalities in Equation (8). A stronger protocol allocates explicit regression budgets to historical task groups and assesses them with appropriate uncertainty control. It must evaluate the deployed composition: independently successful edits can interact after merging, and variant isolation still depends on the behavior of the shared router.

From empirical selection to confidence-adjusted acceptance. Let $\bar{R}_k(H) = k^{-1} \sum_{s=1}^k R_s(H)$ be risk on the uniform historical mixture. Suppose round k constructs at most K candidate harnesses before accessing a fresh validation set V_k of m independent tasks from this mixture. Conditional on the preceding history and the constructed candidates, V_k is independent of candidate construction. For candidate H' , define

$$\begin{aligned} \Delta_k(H') &= \bar{R}_k(H') - \bar{R}_k(H_{k-1}), \\ \hat{\Delta}_k(H') &= \frac{1}{m} \sum_{z \in V_k} [\ell(H', z) - \ell(H_{k-1}, z)]. \end{aligned} \quad (9)$$

Each paired difference lies in $[-1, 1]$. Hoeffding’s inequality [126], conditional on the history, and a union bound over at most KT comparisons, where T is the number of rounds, imply that, with probability at least $1 - \delta$, simultaneously for every evaluated candidate and round,

$$\Delta_k(H') \leq \hat{\Delta}_k(H') + \sqrt{\frac{2 \ln(KT/\delta)}{m}}. \quad (10)$$

Accepting a candidate only when the right-hand side is negative therefore certifies improvement on that round’s mixture, under these assumptions; otherwise the incumbent can be retained. In the terms of Section 6.1, this is the improvement claim with its three indices set: the reporting tasks are fresh draws from the uniform historical mixture, the baseline is the incumbent H_{k-1} , and the round is the one at which the candidate is accepted. Here the same tasks V_k decide

admission and report the gain, which the selection condition rules out for an uncorrected score; the union bound over KT comparisons is what makes this use valid, and reusing V_k without that correction, or in a later round, would not be. This sufficient condition is conservative, but makes explicit why a positive empirical gain alone is insufficient. The guarantee applies to the evaluated candidate, not to an untested merge or a further revision made after inspecting V_k . Nor does it imply improvement on every historical distribution, on an arbitrary future distribution, or at any later round.

HarnessBank provides a concrete step toward uncertainty-aware admission through task-level paired comparisons and a significance gate [97]. However, its screening tasks are sampled from the training set, and its stated per-comparison threshold does not establish the fresh-data and simultaneous error control assumed in Equation (10). SkillOpt and SkillOpt-Lite gate updates using validation gains [49, 50], while MCE stores validation results in its evolutionary archive [51]. Such feedback makes admission adaptive. Similarly, Self-Harness’s fixed held-out set influences later candidates through repeated promotion decisions, even though its task contents are hidden from the proposer. Reusing a holdout in this way requires an adaptive-valid analysis or a controlled reuse mechanism [127]; a union bound alone does not restore the missing independence.

Lack of a per-round certificate does not invalidate evaluation on a separate, untouched final test set. HarnessBank and MCE’s offline protocol report such a separation, and SkillForge reserves a final temporal partition for evaluation [46, 51, 97]. These protocols provide evidence about the final system under the evaluated test conditions, rather than certifying every accepted update. Temporal tests additionally require care when interpreting results under distribution shift.

What these guarantees measure. The independent unit is a task, not an individual rollout: repeated executions of the same task can reduce execution noise but do not create new independent tasks. Equation (10) also holds for unbiased bounded estimates of each paired loss when task clusters are independent. Correlated user sessions or support tickets require corresponding grouping or a dependence-aware analysis, and non-additive metrics require their own concentration arguments. Finally, validation only assesses the specified loss. Fresh tasks do not prevent exploitation of a misspecified evaluator. If evaluator risk differs from target risk by at most η for both a candidate and its incumbent, the upper bound on their target-risk difference requires an additional 2η term. Estimating or bounding this discrepancy requires external evidence, such as executable checks or independent human review. Overall, reliable evolution needs reusable updates, historical coverage, and selection-aware evaluation; none alone establishes all three.

6 Evaluating Harness Self-Evolution

Evaluation serves two roles in harness self-evolution: it ranks and admits candidates inside the update loop, and it supplies evidence about the resulting system. Using the same feedback for both can make a selected gain look like an independently measured one. Even with separate reporting tasks, a comparison must account for how the harness changes execution: recent studies find substantial performance differences under a fixed model and show that published comparisons often leave the harness uncontrolled [128–130]. For continued adaptation, evaluation must also follow what happens to a gain as later tasks and updates change the harness.

6.1 Four Claims Behind a Task Score

A reported gain raises questions about both the evidence used to estimate it and the mechanism credited with producing it. We distinguish selection, activation, attribution, and improvement, summarized in Figure 5.

Four Claims Behind a Task Score

- **Selection:** The reported number is an unselected estimate, or its inference accounts for selection. For an uncorrected estimate, no outcome on the reporting tasks fed the proposal, the ranking, the admission, or the choice of which round and configuration to report.
- **Activation:** The accepted update operated in the runs that produced the number. It is read from execution records, not scores, at three grades: available, reached, and carried.
- **Attribution:** The difference came from the update, not from execution variation, a larger inference envelope, or a changed evaluator. It needs arms that differ only in the update, a controlled envelope, and a margin beyond measured execution variation.
- **Improvement:** The difference holds away from the conditions that produced it. Three indices locate it: distance from the tasks that formed the update, baseline, and round.

The required evidence depends on the claim. Crediting a gain to a particular accepted component requires activation and attribution; extending that gain beyond the tasks that formed the update also requires improvement. A frozen final-state comparison can estimate overall performance against a baseline without tracing individual components. Selection and attribution determine whether a positive difference can be credited to the evolved state under the tested conditions; the improvement claim specifies how far that difference extends. Evidence about the final state does not certify every update on the path to it.

Task distance describes how reporting tasks relate to those that formed the update: held-out tasks test generalization, while later probes on earlier distributions test retention (Section 5.2). Selection asks how their outcomes were used. A held-out score that ranks candidates or decides admission is search feedback; using it to report a gain requires inference that accounts for selection, as in Section 5.2.

Persistence establishes that an accepted state was saved and loaded by a later run (Section 2); activation records what happened after loading. A controlled component ablation with an effect beyond measured execution variation implies that the component operated somewhere, but does not identify the runs or paths involved. Activation is therefore assessed from direct execution records, with the outcome difference considered separately under attribution.

6.2 Current Evaluation Practices and Benchmarks

Corpus and evidence coverage. We audit the *primary comparison* of each method in the two update-target branches of Figure 3: its headline result against a same-model baseline or pre-update state. The curated corpus covers work through 8 October 2026, with 93 scored comparisons and 2 methods without empirical evaluation; five additional works read for the audit fall outside its scope. All four verdicts refer to the same comparison; other experiments are recorded separately.¹

Figure 6 summarizes evidence coverage, and Table 4 gives the per-method verdicts. Activation requires direct records of at least one identified update reached or carried in scored runs, with coverage recorded separately. Improvement is graded on task distance only; it does not certify a positive effect or its persistence. Selection is graded on the uncorrected estimates reports provide. These criteria describe evidence, and their combined count is not a method ranking: a final-state comparison need not trace individual components.

¹[Audit materials](#) provide the rubric, source passages, and coding history.

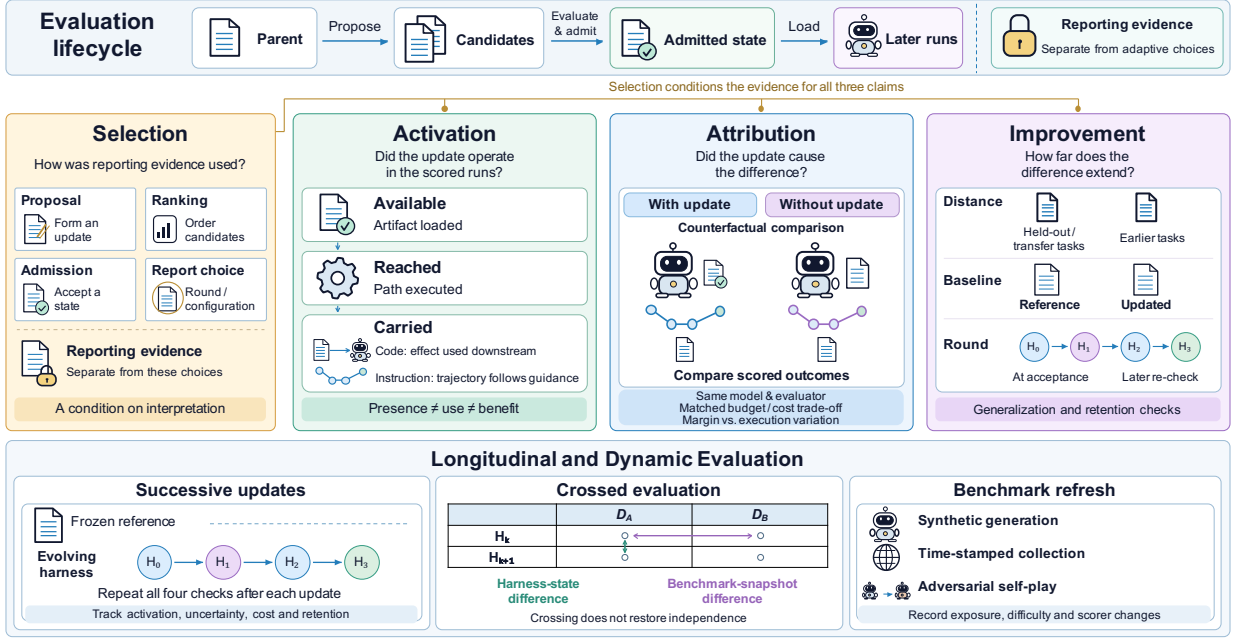


Figure 5 Four claims behind a reported gain. Selection concerns how reporting evidence was used; activation, attribution, and improvement concern the update’s operation, causal effect, and reach. The lower panel extends these questions across successive updates and separates harness changes from benchmark refresh. Table 5 specifies the corresponding controls.

The protocols differ in where their evidence enters the loop. HarnessBank separates final scoring from search, but logs its patch-activation beacons during screening [97]. Self-Harness tests candidates on tasks hidden from the proposer, yet uses those outcomes for promotion [87]. AHE replaces individual layers to examine their contributions, without reporting uncertainty for the resulting margins [90]. These cases motivate the distinctions below: a check can support admission without establishing final-run activation, independent evaluation, or a lasting gain.

Several recent benchmarks build parts of these claims into their protocol. HarnessOpt-Bench keeps its test partition out of reach for the whole search and meters the resources the target agent spends, which covers selection and the inference envelope. HarnessDev and GDPevo score evolved harnesses on held-out tasks. GDPevo builds those tasks by recombining business rules spread across its training tasks, to test whether training experience transfers to new combinations. EvoHarnessBench grows the harness an agent is given and measures whether earlier competence is retained [131–134].

6.2.1 Evidence for Selection

A task score can guide a proposal, rank candidates, decide admission, or estimate performance. If the outcomes used for reporting also selected the candidate or the reported round, the uncorrected gain may overestimate the effect. Selection therefore concerns the history of the reporting evidence; calling a split “validation” or “held-out” does not establish its independence.

Reporting evidence is used in candidate generation when the reporting tasks generate the update. This is the case most specific to self-evolution. A loop that reads the failures of its scored runs and writes a repair from them has used those tasks as a training signal, whatever the split is called and however the candidate is ranked later. SWE-Exp builds its experience bank from search runs on the same 500 SWE-bench Verified issues it then scores [40], and MOSS feeds four tasks into evolution and re-scores the candidate on the same four [108]. No ranking rule or stopping rule repairs this, because the reporting evidence was already used to generate the candidate. Two protocols look similar but keep scoring separate from feedback. AWM scores each WebArena task

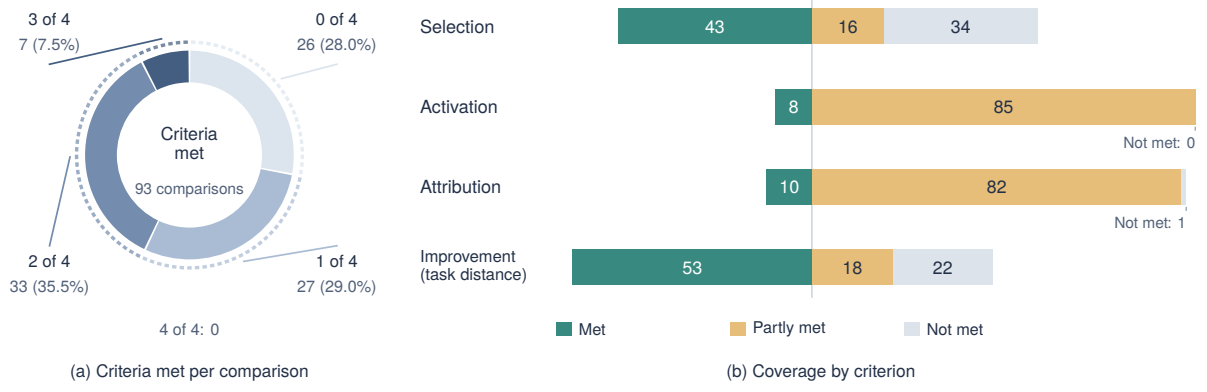


Figure 6 Evidence coverage across 93 primary comparisons. (a) Number of criteria met; each outer tick represents one comparison. Partial evidence can remain when none is fully met. (b) Coverage by criterion; every row totals 93. Activation records existence with stated coverage; improvement grades task distance. Counts describe reporting evidence, not overall method quality.

before its outcome may enter the workflow memory, so no outcome reaches the update before it is scored [35]. Such a number is a valid online measurement of the learning policy, rather than of one frozen final state; what it lacks is distance from the stream, which the task-distance criterion grades. Reflexion adapts inside a task from self-generated tests and never sees the benchmark’s verdict [8]. Its number is also a measurement, but of adaptation within one task, not of an update carried to new tasks.

Reporting evidence is used for ranking when the reporting partition also orders the candidates. Offline prompt and workflow search is designed to avoid this: it optimizes on task feedback, ranks on validation, and reports on a partition touched by neither [9, 25]. Once the reporting partition ranks, prunes, or guides candidates, the reported score is the best of the pool on that partition. OPRO shows the instruction with the highest test accuracy, and ProTeGi takes its test figures from the best of a final beam scored on the test split [22, 23].

Reporting evidence is used for admission when a held-out score enters the promotion rule. This is harder to see in a persistent loop, because a task set can be hidden from the proposer and still shape later candidates through repeated admission decisions. Self-Harness promotes an edit only when its pass count does not fall on either a held-in or a held-out split and rises on at least one, and it never shows the held-out split to the proposer [87]. The held-out pass count is nevertheless half of the promotion rule and is consulted every round, so the split is not independent of the search. The gate tests regressions on tasks withheld from proposal generation; it does not supply an independent final estimate [135, 136]. Keeping this gate while evaluating the frozen promoted state on an additional untouched partition would preserve its admission role and supply separate evidence about the state reached.

Reporting evidence is used to choose what to report when the reported round or configuration is picked on the reporting tasks. HarnessX reports peak accuracy on its evolution tasks and says itself that the numbers carry selection bias and potential overfitting, and its own trajectories show how far a peak can sit above the final state: one configuration peaks at 73.8% and ends its run at 49.5%. AHE carries forward the round with the highest pass rate on the tasks it evolved against [90, 116]. Reporting the maximum of a trajectory, with no stopping rule fixed in advance, turns the reporting metric into an objective of the search, and the same happens when a configuration is chosen instead of a round: SkillOpt’s headline row matches the best rows of sensitivity sweeps run on its reporting benchmarks rather than its stated default, EGO-Prompt reports the best of three repeated runs, and APE reaches human level on all 24 instruction-induction tasks after retuning its hyperparameters and re-running only the five test tasks an earlier version had fallen short on [21, 28, 49].

Table 4 Evidence-criterion coverage in each method’s primary comparison. ✓ met; ◦ partly met; — not met, based on the primary source and the criteria in Section 6.2. Act. requires an identified scored-run instance; Imp. grades task distance. Other experiments are not pooled. Cordis and DeepSeek Harness report no empirical evaluation and are omitted.

Method	Sel.	Act.	Att.	Imp.	Method	Sel.	Act.	Att.	Imp.
ACE [48]	◦	◦	◦	✓	Mass [75]	✓	◦	✓	✓
ADAS [100]	◦	◦	◦	✓	MCE [51]	✓	◦	◦	✓
AFlow [9]	✓	◦	◦	✓	MemCoder [41]	◦	◦	◦	◦
Agent KB [38]	—	◦	◦	◦	MemEvolve [61]	✓	◦	◦	✓
AgentDevel [106]	◦	◦	◦	✓	Memory Management [68]	✓	✓	◦	◦
AgentSquare [78]	—	◦	◦	—	Memory Transfer [43]	—	✓	◦	✓
AHE [90]	—	✓	◦	—	MemP [34]	◦	◦	◦	✓
APE [21]	—	◦	◦	✓	Meta-Harness [101]	✓	◦	◦	✓
Argus [117]	—	◦	◦	—	Meta-Team [83]	✓	◦	◦	✓
Autogenesis [118]	✓	◦	◦	◦	metaTextGrad [27]	✓	◦	◦	✓
AutoSaddler [91]	✓	◦	◦	✓	MIPRO [20]	✓	◦	✓	✓
AWM [35]	✓	◦	◦	◦	ModularRSI [119]	✓	◦	◦	✓
CHASE [93]	✓	◦	◦	✓	MOSS [108]	—	◦	◦	—
CodeSkill [45]	◦	◦	◦	◦	MUSE-Autoskill [56]	—	✓	◦	—
DarwinX [103]	—	◦	◦	—	OPRO [22]	—	◦	◦	✓
DEEVO [31]	◦	◦	✓	✓	Ouroboros [107]	◦	◦	◦	◦
DelvePO [30]	◦	◦	✓	✓	PolySkill [84]	—	◦	◦	◦
DGM [19]	—	◦	◦	◦	Promptbreeder [18]	✓	◦	◦	✓
DSPy [76]	—	◦	◦	✓	ProTeGi [23]	—	◦	◦	✓
Dynamic Cheatsheet [47]	—	◦	◦	◦	ReasoningBank [39]	✓	◦	◦	◦
Ecdysis [94]	✓	◦	◦	✓	ReCreate [98]	✓	◦	◦	✓
EffiSkill [59]	✓	◦	✓	✓	Reflexion [8]	✓	◦	◦	—
EGO-Prompt [28]	—	◦	◦	✓	RHI [82]	—	◦	◦	—
Evo-Harness [92]	✓	◦	◦	◦	RoboPhD [99]	✓	◦	◦	✓
EvolveNet [109]	✓	◦	◦	✓	RRSI [102]	✓	◦	✓	✓
EvoRepair [42]	—	◦	◦	—	Self-Harness [87]	◦	✓	◦	◦
ExpeL [6]	✓	◦	✓	✓	SePO [29]	✓	◦	◦	✓
FlowEvo [85]	✓	✓	✓	◦	SEW [80]	◦	◦	◦	✓
Gödel Agent [111]	◦	◦	◦	✓	SICA [10]	—	◦	◦	—
GEPA [25]	✓	◦	◦	✓	Skill Consumption [67]	✓	◦	◦	✓
GPTSwarm [74]	—	◦	◦	—	SkillClaw [37]	—	◦	◦	—
Group-Evolving Agents [105]	—	◦	◦	◦	SkillForge [46]	✓	◦	◦	✓
GSE [55]	✓	◦	◦	✓	SkillMOO [58]	—	◦	✓	—
GSkill [44]	✓	◦	◦	✓	SkillOpt [49]	—	◦	◦	✓
GUI-HARVEST [95]	◦	◦	◦	◦	SkillOpt-Lite [50]	✓	◦	◦	✓
Harness Ceiling [96]	✓	◦	◦	✓	SWE-Exp [40]	—	◦	◦	◦
HarnessBank [97]	✓	◦	◦	✓	Symbolic Learning [79]	—	◦	—	—
HarnessCompass [89]	—	◦	◦	—	TACO [57]	—	◦	◦	—
HarnessFix [88]	✓	◦	◦	✓	TextGrad [24]	✓	◦	◦	✓
HarnessX [116]	—	✓	◦	—	Trace [26]	✓	◦	◦	✓
HGM [104]	—	◦	◦	—	Trace2Skill [36]	✓	◦	✓	✓
Janus [54]	—	◦	◦	✓	TTHE [112]	—	◦	◦	—
JudgeFlow [81]	◦	◦	◦	✓	VALVE [53]	✓	◦	◦	✓
Life-Harness [113]	✓	◦	◦	✓	VASO [60]	—	◦	◦	—
Live-SWE-agent [86]	✓	◦	◦	—	VERSE [110]	◦	◦	◦	✓
Living-Harness [52]	—	◦	◦	—	Voyager [7]	✓	◦	◦	◦
MaAS [77]	◦	✓	◦	✓					

Sel. selection Act. activation Att. attribution Imp. improvement (task distance).

A separate failure occurs when the candidate changes the record used to judge it. Studies of this failure report patches based on faults that were never observed in execution, direct edits to verification logic, unreliable self-authored checks, and, in SWE-bench, leaked solutions and weak assertions [137–140]. An apparent gain then reflects a changed measurement, not a better harness, and the remedy is structural. HarnessBank keeps its evaluation and bookkeeping code in a kernel that candidates cannot modify, and HarnessX applies the reference test patch before the model patch and blocks writes to test files [97, 116].

The remedies for these four forms of evidence reuse are procedural. Protocols reserve reporting evidence from proposal, ranking, and admission, and either specify the reported round and stopping rule in advance or account for their adaptive choice. They treat repeated data-dependent comparisons with anytime-valid procedures [141, 142]. They also collect tasks after a stated cutoff, although a later release date prevents neither adaptive reuse nor shifts in difficulty [143, 144].

6.2.2 Evidence for Activation

A skill can be loaded into context and ignored, and executable code can be installed without its path ever being taken. Evidence that an update operated comes at three grades of increasing strength. None needs a counterfactual run, and each counts only when recorded in the scored runs.

The weakest grade, available, shows that the artifact entered the run. SkillsBench records whether task-provided skills are read or referenced, Voyager measures whether synthesized programs are retrieved, and Lin et al. report a skill-load rate over trajectories [7, 70, 71]. An artifact the agent never acts on also satisfies it, which is why availability alone does not establish use.

The second grade, reached, shows that the changed path executed. Several systems can record it, but most do so outside the scored runs. Symbolic Learning stores each node’s inputs, outputs, and tool calls along the executed graph, as the trajectory its training loss is computed from [79]. AutoSaddler records a trace diff before and after each patch in its optimizer’s diagnosis sessions [91]. HarnessBank requires each candidate to declare a beacon that fires when the patched path triggers, and rejects during screening any candidate whose beacon never fires [97]. Its beacons are logged on the screening subset, and none is reported for the sealed test behind its headline result. The protocol therefore separates final scoring from search without reporting whether identified edits operated in those runs. Recording the same beacons during the frozen sealed-test evaluation, without feeding them back into candidate selection, would connect the mechanism to that comparison.

HarnessX records this grade in the runs it reports. Each edit declares the trace feature that must appear if the edit fired, and the declaration is checked against the next round’s traces, which on its evolution tasks are the reported runs. On the GAIA / Sonnet 4.6 configuration, whose reported peak is round 11 at 83.5%, a round-10 composite edit declares a call to a named retrieval tool, and the trace analysis behind the reported round finds that the tool fixed retrieval for most newly passing tasks [116]. The record covers one of 15 configurations and shows that the edit fired, not how often across them. HarnessX calls this field an attribution signature. Under our reading it is activation evidence, and attribution (Section 6.2.3) is a further claim. AHE shows, in rollouts of the reported round, the token that marks its submit guard as engaged and the shell-layer refusals that name the protected paths, and Self-Harness pairs its headline run with traces in which the retained guard shapes the trajectory [87, 90]. Both show cases, not rates over the scored runs.

The third grade, carried, records that the changed path’s effect reached its intended downstream use. For code supplying information to the model, the output must survive into the model’s input. HarnessX requires a code-bearing candidate to show, before it ships, that its result remains present after prompt assembly [116]. This catches a component whose output is dropped or truncated after its execution beacon fires. Runtime changes can also act through control decisions or external outputs: a stopping rule may prevent another step, a dispatcher may select a different

implementation, or a tool may write the final artifact. In these cases, a downstream control or output record connects the reached path to what execution subsequently did; entry into the model’s context is not the relevant endpoint.

For an instruction or skill, carried evidence records adherence in the trajectory. Lin et al. score trajectories against a rubric extracted from the skill text and report adherence separately from loading. Two agents in their evaluation load skills at almost the same rate, 0.961 and 0.957, while their trajectories follow the loaded skill at 0.350 and 0.757, with adherence for the weaker agent falling from 0.52 at loading to 0.13 at the final turn [71]. Output survival and adherence thus record different downstream uses of an artifact. Both help locate where its influence can enter execution; attributing an outcome gain requires the controlled comparisons in Section 6.2.3.

The three grades correspond to three rates a report can state next to its task score. For an update u , let $R(u)$ be the scored executions satisfying a declared applicability rule for u , specified independently of whether it fired or succeeded. Within $R(u)$, let $L(u)$ be those that load it, $B(u)$ those in which its declared beacon fires, and $C(u)$ those in which its effect carried. Then

$$\begin{aligned} \text{load}(u) &= \frac{|L(u)|}{|R(u)|}, & \text{reach}(u) &= \frac{|B(u)|}{|R(u)|}, \\ \text{carry}(u) &= \begin{cases} |C(u)|/|L(u)|, & \text{interpreted guidance,} \\ |C(u)|/|B(u)|, & \text{executable paths.} \end{cases} \end{aligned} \quad (11)$$

For an instruction or skill, carry is the adherence rate Lin et al. report. For an executable path, carry is conditioned on reaching that path. Restricting $R(u)$ to executions where the update applies keeps a component intended for rare tasks from looking inert; conditioning carry on entry separates downstream use from how often the artifact was loaded or reached. A rate is undefined when its denominator is zero; reports should state the corresponding counts rather than replace it with zero. Current instruments report less than this. HarnessBank’s gate asks only whether the beacon fires on any task or attempt, so it separates a candidate that never fires from one that fires once, but not one that fires once from one that fires throughout [97]. Lin et al.’s headline rate is per trajectory, which hides partial and time-varying adherence; their instruction-level judgments and phase scores recover part of it [71].

Loading policy can also change how much of a revision’s behavioral effect reaches later execution. In a separate sandbox study across four agents, a rule revision raises the required-action rate by +0.23 when the skill is preloaded, but retains about 51% of that action gain on average when the skill body is available only on demand [145]. This controlled behavioral comparison illustrates why recording availability and use is informative even when the revised text is unchanged.

6.2.3 Evidence for Attribution

Execution variation, additional inference effort, and evaluator changes can each affect the observed margin. Attribution requires accounting for these factors in the comparison and, for a component-level claim, isolating the component through a counterfactual run.

Execution variation. Execution variation is estimated by re-running a frozen state on the same tasks; repeated runs estimate noise and do not create new independent tasks. The observed gain must be compared with the variation of the unchanged state, with the unit of replication stated. Existing reports provide different levels of evidence. DarwinX averages five runs and states the spread on its main benchmark, adding a paired McNemar test on its held-out split, whose $p = 0.45$ it calls suggestive [103]. TACO uses five runs with matched budgets and reports standard deviations that put its 1.4-point headline margin at about one standard error of the difference, and AutoSaddler reports the mean and standard deviation of three runs [57, 91]. HarnessBank averages

three attempts per task and screens candidates against the spread of the paired differences, not against zero [97]. AHE, Self-Harness, and MOSS run two or three executions per task and report neither intervals nor paired tests [87, 90, 108]. Spread over judges or evaluator seeds does not meet the requirement: RHI averages two judges and three seeds while each iteration executes the agent once, so its spread measures the evaluator, not the execution the update depends on [82]. Neither does spread belonging to a different comparison: ProTeGi repeats its optimization twelve times but reports the resulting standard error against a competing optimizer at its own budget, while the headline sets the evolved prompt against the original one, and the main results average three trials without a spread [23].

A large margin is not a substitute for measured variation. Bounds computed from task counts depend on assumptions about task independence and covariance between arms, while dispersion from another model, split, or configuration need not describe the primary comparison. We therefore credit this condition only when repeated runs, a paired test, or reported raw data distinguish the primary margin from zero under the stated rubric. Self-Harness reports a large AppWorld gain but no interval or test, so Table 4 records partial attribution [87]. Similarly, Harness Evolution Hits a Ceiling reports a held-out gain from 0.164 to 0.297, but the headline pair uses four versus two rollouts on different nights. Dispersion pooled across other configurations does not establish uncertainty for this pair, so its attribution criterion is only partly met [96].

Inference envelope. The inference envelope is the compute spent in the runs being compared. Workflow expansion, retries, critic or verifier calls, extra tools, and larger candidate pools can all raise the final score. A controlled comparison can estimate the total effect of an update, including the resources it induces. The stricter attribution criterion used here asks whether the gain survives control for additional inference effort; it does not treat every total-effect estimate as invalid. A comparison can handle this by matching the envelope, holding the inference budget fixed across arms; by adjusting for cost, reporting performance together with resource use or as a performance–cost frontier; or by declaring the envelope, letting the budget change and reporting the change. Only the first two control it, since a declared change tells the reader that the budget may explain the gain, not that it does not. Current practice covers all three. TACO matches budgets across arms, Meta-Harness and MCE plot accuracy against context tokens as a frontier, and HarnessX declares per-task inference cost with its gains, which falls by a quarter on one benchmark and rises by sixty percent on another [51, 57, 101, 116].

Two budgets are easily confused. Search cost, the rollouts and tokens spent finding the update, belongs to the method; the envelope is the inference compute of the scored runs, and only it can explain a difference between them. HarnessFix reports 57.6M tokens of offline repair cost, and HarnessBank keeps its search within 2.1 times that of its baselines. Neither figure says what the scored runs consumed, and the credited repairs in both add verification or finalization steps inside the agent loop [88, 97]. DarwinX shows the cost of an envelope that is declared and not matched: its matched-model pair runs the unevolved baseline at the provider’s default reasoning effort and the evolved harness at high, and on the six tasks it newly solves its own figure shows the evolved harness taking twice the turns and four times the tokens. The reported margin is therefore the joint effect of the new logic and a higher effort setting, and its comparison with Terminus-2 at xhigh effort does not separate the two, because Terminus-2 differs from the base harness in more than its budget [103].

Two envelope changes are specific to self-evolution and easy to miss, because neither appears as an edit to a budget setting: an update that adds retries or verifier calls inside the agent loop raises the number of model calls per task without changing any declared limit, and an update that accumulates skills, instructions, or memory lengthens the context, raising the tokens processed on every later task. The first is compute the unchanged state could also have been given, so it has to be matched or traded off before the gain is credited to the logic; the second is the artifact’s own

context overhead. This audit records that overhead as part of the update rather than a separate competing cause, while requiring it to be declared. A fixed-total-token comparison can additionally test whether the gain survives a common resource cap. No single cost measure fits every harness, and the declared envelope should cover whichever resources the update changed [146, 147].

Evaluator. The evaluator is the third confound: the score can move because the measurement moved. With reward inflation or a co-evolving grader, a higher score may reflect a change in the stored evidence or in the judge, not in the harness [148, 149]. The corresponding checks are meta-evaluation against expert annotation and detection of judge drift [150, 151]. HarnessBank reports that on five domains the mechanism its gate credits matches the dominant failure pathology, across domains scored by deterministic verifiers and domains scored by a judge, which argues against a scoring artifact without re-scoring any result under a second evaluator [97].

Counterfactual runs. A counterfactual run changes one component and holds the rest of the run as stable as practical. AHE swaps each evolved layer into its seed alone and reports each swap as a single number on 89 tasks [90]. These substitutions ask what a layer contributes to the seed. Removing that layer from the fully evolved harness would ask a different question, its contribution in the presence of the other evolved layers; interactions can make the answers differ. Repeated paired executions under controlled inference budgets would address uncertainty in either comparison, whereas the reported single margins do not establish whether differences among layer scores exceed noise. CHASE instead builds the counterfactual into admission, admitting a patch only when its gain over the parent survives every benchmark transformation a challenger has confirmed, so the gain is tested against a change of protocol and not only against noise [93].

Two systems go further and check each shipped edit against the next round’s task-level outcomes. HarnessX records, for each edit, the tasks it should unlock, stabilize, or put at risk, and checks them against the following round [116]. AHE does the same and reports how accurate its predictions were: for the tasks an edit would fix, precision and recall are roughly a third and a half, while for the tasks it would break both stay near a tenth, about twice the random baseline [90]. These records test whether predicted task-level changes materialize. Their asymmetry suggests that anticipating regressions is harder than anticipating repairs in this setting; without controlled comparisons, they do not establish which changes the edit caused. Trace-level checks read the same evidence for a different question, whether a newly passed task used the intended mechanism or exploited the evaluator: HarnessX reports three cases its authors file under reward hacking, including tasks passing by matching format regularities in the verifier, one benchmark’s mean reward rising while its exact-match rate stayed fixed, and candidates clearing every gate while their score fell, because the gates check replay, novelty, and structure but not pass rate [116]. All of this evidence is local to the evaluated tasks and configurations; whether the difference extends beyond them is the improvement claim.

6.2.4 Evidence for Improvement

Current evaluations test the reach of an update through local failure repair, frozen transfer, or an online trajectory. These protocols differ in which tasks they cover and when they measure the evolving state.

Local failure repair asks whether a candidate can re-solve the task that produced the update. MOSS checks its repairs with execution and recovery evidence on the same four tasks that formed the update batch [108, 152]. Repair shows that the system operates, but it does not meet the task-distance criterion, because the originating task has already shaped the candidate and because re-running only earlier failures benefits from regression to the mean. A stronger repair claim compares the candidate with its unchanged parent under matched repeated executions and includes

tasks that previously passed.

Frozen transfer evaluates a selected state without further updates. AHE transfers a harness evolved on Terminal-Bench 2 to SWE-bench Verified, and Voyager and ExpeL test reuse on new tasks, environments, or domains [6, 7, 90]. AHE’s seed scores 75.2% on the 500 SWE-bench Verified issues and its frozen evolved harness scores 75.6%; the report does not test the uncertainty of this 0.4-point difference. This secondary transfer result is separate from AHE’s in-domain Terminal-Bench 2 headline, which determines its verdict in Table 4.

Online trajectories report outcomes while the harness changes along a task stream. ACE varies task order, and SICA evaluates successive code mutations with latency and token use in its utility [10, 48]. A rising trajectory may depend on task order, conceal regressions on earlier tasks, or reflect growth in context and inference effort. A frozen control and periodic retention probes help separate these effects from gains that persist beyond the current stream.

6.3 Longitudinal and Dynamic Evaluation

Longitudinal evaluation follows an update beyond its initial comparison. A claim that each accepted revision helps requires evidence at each transition; a claim that an acquired capability lasts requires later probes. Both face two changing quantities: the harness state and the task stream or benchmark. The protocols below establish references for these comparisons, track retained state, and separate harness changes from benchmark refresh.

6.3.1 Evaluating Successive Updates

When updates follow one another, each transition $H_k \rightarrow H_{k+1}$ depends on earlier rollouts, feedback, and admission decisions. Outcomes along this history are therefore not fresh, independent evidence for each later state, and crediting a change to a particular update needs references that do not move with the agent. The first such reference is the model, which Section 2 assumes fixed. Holding M fixed is a property of the study, not of the method. A hosted model may be revised, re-quantized, or re-routed between rounds with no change on the harness side, and a comparison that spans the change credits H_{k+1} with a difference M produced. Pinning a provider version and recording it with the decoding parameters makes the assumption checkable. Where no version can be pinned, the model is an uncontrolled input and the report should say so. Table 5 lists seven controls with the claim each protects and the confound it addresses.

Historical probes must remain outside the update loop to supply independent retention evidence. Once their results guide later revisions, their reuse requires the adaptive-valid treatment discussed in Section 5.2. Task-order permutations complement these probes by measuring sensitivity to the sequence of experience.

6.3.2 Retained State Across Rounds

The state that carries a learned procedure may change during maintenance. MUSE-Autoskill, for example, can merge overlapping skills and prune failing or unused entries [56]. A procedure may remain useful in a revised or merged artifact after its original form is removed. Conversely, an unchanged skill may become harder to retrieve or less likely to be followed as the surrounding library and runtime change. Later-task probes assess whether the capability remains; lineage and execution records help identify what now supports it. A claim about the continuing contribution of a particular update additionally needs a comparison that isolates it within the later harness, under the controls of Table 5.

Several suites evaluate state carried across task sequences or linked sessions in which earlier writes affect later work [155–159]. Recording that state alongside outcomes makes the learning mechanism easier to inspect. ServeLearnBench streams environments whose undisclosed rules change over time; across learning harnesses and models, it finds that continual adaptation incurs

Table 5 Controls for evaluating successive updates. Rows are grouped by the claim each control protects. The second column specifies the records needed to check each control, and the third identifies the confound it addresses. Citations point to an instance of the control or to evidence that the confound occurs.

Control	What to record	Confound addressed
<i>Selection: was the number measured or picked?</i>		
Evidence separation [131, 141]	Which tasks and signals were used for proposal, ranking, admission, and reporting, and how the reported round and configuration were chosen.	Reporting evidence reused by the search.
Oracle integrity [97, 138, 139]	That scoring, verification, and bookkeeping code lie outside what a candidate can edit or repeatedly probe.	A candidate changing the measurement that judges it.
<i>Activation: did the update operate in the scored runs?</i>		
Activation record [71, 90, 116]	For each task and attempt in the scored runs, the three rates of Equation (11): the artifact was available, its path ran, and its effect carried.	An update credited although it never operated.
<i>Attribution: did the difference come from the update?</i>		
Execution variation [5, 103, 146]	Repeated runs of the frozen state on the same tasks, or a paired test over per-task outcomes, with the margin set against that spread. Spread over judges or evaluator seeds does not count.	Noise read as progress.
Inference envelope [103, 116, 146, 147]	For the scored runs: generated tokens, model and verifier calls, turns, retries, attempts or candidates sampled, and the reasoning-effort setting. Context added by the artifact and search cost are stated separately.	Added inference compute read as a harness gain.
Model and evaluator record [146, 149, 153]	Model identifier and version, decoding parameters, scorer and judge versions, and any change to them during the study.	A changed model or grader read as a harness gain.
<i>Improvement: how far does the difference extend?</i>		
Controlled histories [154]	Varied task orders, a frozen reference state kept alongside the loop, probes on earlier distributions the loop does not control, and a re-measurement of each credited gain several rounds later.	Order effects read as transfer or retention; a gain measured once read as lasting.

costs and can degrade behavior that was already correct [160].

Work on retirement, voting-based curation, library drift, and contaminated or self-poisoning updates shows how later revisions change which artifacts remain available and which get selected [161–165]. An activation record at admission therefore needs later checks of retrieval, execution, and adherence. Rollback also requires checking which state resumes operation: neither AHE’s file-level reverts nor MOSS’s image rollback on failed health probes reports whether the reverted state is the one later runs load [90, 108].

Other suites score the artifacts a loop produces, or lengthen the horizon over which it is observed [166–170]. The count, validity, or diversity of artifacts is evidence that state was produced, not that it was activated or that it improved anything, and a long horizon by itself measures neither forgetting nor accumulated cost.

PAST-Bench and FinEvo-Bench compare runs with persistence enabled and disabled [171, 172]. This intervention estimates the effect of the persistence mechanism as a whole. Isolating a particular stored item’s contribution requires a more local intervention, with execution records establishing where the item operated.

Several recent studies measure retained performance directly. VALVE freezes its current bank at step 0 and every ten mixed tasks, scores each frozen checkpoint on the same disjoint 100-task

panel with three rollouts per task, and reports peak-to-final drawdown. The peak is selected on that panel, so it is not an untouched final-test estimate [53]. PolySkill revisits its starting WebArena Shopping tasks after adapting to Amazon and Target and ends +4.9% over ASI, in a case study separate from its primary online comparison [84]. Both track the later performance of an evolving state.

Generalization measurements and admission gates answer related but different questions. Agent Plasticity measures how much experience an agent converts into held-out gains [173]; a study of 21 skills that improved their training tasks finds that five retain all of the improvement on held-out tasks, thirteen retain part, and three retain none [174]. Self-Harness requires non-regression on its current evaluation splits, whereas HarnessX checks replay, novelty, and structure without guaranteeing non-decreasing task scores; neither establishes retention of capabilities outside the tested conditions [87, 116]. HarnessBank’s retention ratio divides test gain by training gain; it describes transfer and is not a statistical test or a measure of performance after later updates [97].

The studies above leave a more specific question unresolved: the later causal contribution of a particular accepted update. AHE notes that one repaired task remains solved through later iterations, but this does not isolate the retained contribution of the edit that first repaired it [90]. Such an estimate requires later probes on an earlier distribution, together with a comparison that isolates the credited update within the subsequent harness state. A longitudinal report should distinguish state carryover, later-run activation, retained performance, and update-specific contribution, while also recording rollback, adaptive reuse of evaluation evidence, and cumulative cost.

6.3.3 Benchmark Refresh and Crossed Evaluation

Refreshing tasks limits saturation and direct contamination. It also introduces a second source of change: candidate updates may be exposed to the refreshed instances, and difficulty may vary between releases. Instance exposure and benchmark version therefore have to be recorded separately from harness updates.

Current benchmarks refresh in three ways, and each leaves a different problem. Synthetic generation derives new instances from existing ones or from diagnosed failures, as in Benchmark Self-Evolving and StressEval, where the generator and the candidate update may read the same traces and the report should record that overlap [175, 176]. Time-stamped collection admits tasks released after a cutoff, as in LiveBench, LiveCodeBench, LiveMedBench, SWE-rebench, and SWE-bench Live; a later release date reduces direct overlap but neither equalizes difficulty across releases nor shows that related task patterns were absent from earlier development [143, 144, 177–179]. Adversarial self-play lets the challenge distribution move with the participants, as in MathDuels and the Red Queen Gödel Machine, so scores depend on the participant pool and the verifier, and changes in the harness, the task generator, and the evaluator must be reported separately [180, 181].

Comparing H_k on snapshot $\mathcal{D}_A$ with H_{k+1} on snapshot $\mathcal{D}_B$ confounds harness and benchmark changes. A crossed comparison evaluates both harness states on both snapshots under the same scoring procedure and controlled inference envelope. Within-snapshot comparisons then estimate the harness difference, while within-state comparisons expose the snapshot difference. Exposure during candidate generation or admission still has to be recorded: crossing the evaluation does not restore independence to evidence already used by the search.

Evaluating every historical harness on every snapshot is usually too costly. A study can instead keep a versioned set of anchor tasks across adjacent snapshots and cross-evaluate selected checkpoints. It should report the resulting coverage and replication, which determine which differences remain detectable.

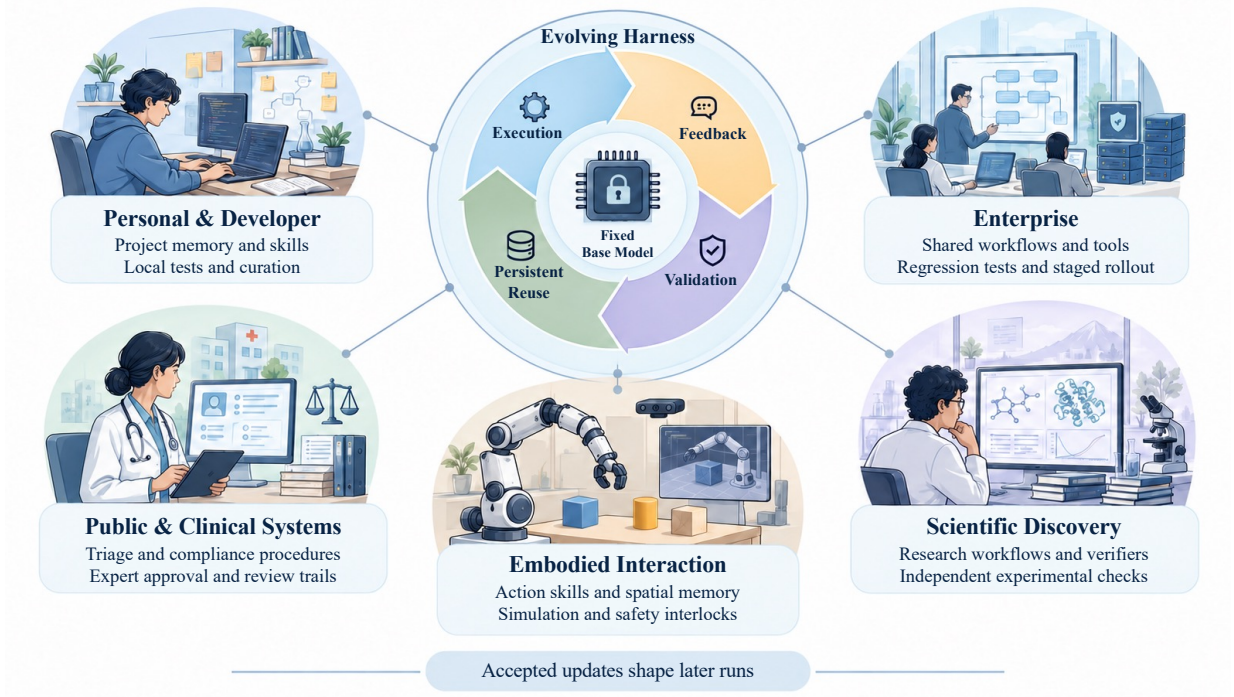


Figure 7 Applications of harness self-evolution across five deployment regimes. With base-model weights held fixed, execution feedback informs harness revisions that are validated, retained, and reused in later runs. The surrounding panels identify representative update targets and validation requirements for personal and developer workspaces, enterprise automation, scientific discovery, robotics and embodied interaction, and public services and clinical systems. These regimes share an update lifecycle but differ in how state is shared, how revisions can be verified, and whether external actions can be undone. The figure summarizes deployment requirements rather than establishing improvement for every application.

7 Applications of Harness Self-Evolution

Harness self-evolution retains project knowledge, reusable procedures, and revised tools for later work. The application determines what this experience must preserve: local conventions in a workspace, shared processes in an enterprise, experimental assumptions in science, or action conditions in robotics. Figure 7 relates these learned artifacts to their deployment settings and the checks needed when they change.

7.1 Personal Assistance and Developer Workspaces

Recurring work creates opportunities to reuse corrections, preferences, project facts, and procedures instead of rediscovering them [12]. Project memories preserve information across sessions in deployed coding agents [62, 182, 183]; file-based skills package procedures that can be invoked when relevant [65, 184–187]. These formats can carry manually authored configuration as well as the algorithmically revised state defined in Section 2.

The maintenance process determines how execution feedback enters this state. Augment and Gemini CLI’s experimental Auto Memory propose memory or skill edits for approval, while Devin recommends prompt and knowledge edits after a session [63, 188, 189]. Prime Agent applies evidence-backed refinements to supplemental prompts, memories, and skill descriptions [190]. These policies give users different roles in selecting which experience becomes standing guidance.

Research systems make the conversion from execution to reusable knowledge more explicit. Dynamic Cheatsheet and ReasoningBank turn self-assessed outcomes into strategies [39, 47]; ACE revises a playbook and Trace2Skill consolidates trajectories into skill directories [36, 48]. SkillOpt and SkillOpt-Lite evaluate revisions before retaining them [49, 50], while Janus compares old and

candidate memories on a small regression set [54]. Some systems report fewer steps per task when retained skills supply previously discovered action or tool-call sequences [191, 192]. These savings come from reusing procedures that earlier runs had to discover.

Keeping that procedure useful requires preserving its scope. A temporary workaround for a dependency version may cease to apply after an upgrade, while a user preference can remain valid across many versions. GitHub Copilot checks repository memories against current code and deletes entries left unused for 28 days [64], illustrating one policy for limiting stale state. Curation must relate revision and expiry to the conditions under which the guidance applies.

Availability is another limit on reuse. An admitted skill may go unloaded or be followed less as a trajectory grows, especially with weaker models [71]; focused skill sets outperform exhaustive bundles, and self-generated skills do not substitute for curated ones [70]. Untrusted inputs can also leave instructions that persist into later sessions [72, 73, 193]. A workspace therefore needs to maintain both the contents of its library and the selection process that brings those contents into later tasks.

7.2 Enterprise Automation

An enterprise update can turn one team’s experience into another team’s operating procedure. Shared skill libraries, triage rules, and repository middleware carry knowledge across people and services as internal APIs, dependencies, and business rules change [11, 12]. SkillForge diagnoses batches of customer and IT support failures and rewrites the deficient skill [46]; SkillClaw pools many users’ trajectories into a shared library [37]. In software engineering, DarwinX, AHE, MOSS, and Ouroboros revise executable tools, middleware, and control logic [90, 103, 107, 108].

Sharing these artifacts also transfers assumptions. A procedure learned by one team may depend on its permissions, data schema, or exception-handling policy. Another team may benefit from the diagnosis while needing different service bindings and authorization. Representing those local requirements separately would make it easier to adapt a shared procedure to its destination. Successful use by the originating team supplies a starting point for this adaptation; cross-team use tests whether the preserved knowledge applies beyond that setting.

Deployment must also preserve the business objective the procedure serves. A shortcut can improve a local proxy while degrading the process it represents [194, 195]. Existing systems use regression limits, patch validation, isolated containers, health checks, and reviewed commits [88, 103, 107, 108]. For shared enterprise state, these checks need to cover the destination teams’ tasks and operational assumptions. The resulting revision can then be assessed as a change to a shared process, with consequences beyond the failures that prompted it.

Rec-RSI studies a related retention problem in recommendation: recommender outputs are used to train successor models [196]. Its marginally matched comparisons of cross- and within-generation pairs use equal inference cost and show that pre- and post-update models can retain complementary ranking decisions. The mutable state is model weights. For enterprise harnesses, this motivates evaluating earlier and revised procedures both separately and in composition before deciding what to retain.

7.3 Scientific Discovery

Scientific campaigns can accumulate procedures across literature search, coding, simulation, and analysis [14, 197, 198]. CORAL retains shared notes, tool settings, and search heuristics during open-ended optimization [199]; Argus maintains memories, workflows, and routing rules in durable campaign state through role-based review [117]. Their inherited state records how research is conducted. AlphaEvolve instead evolves the external programs and mathematical heuristics under study [200], illustrating the adjacent use of evolutionary search on research outputs. PSI’s vision of an AI-native physics laboratory places accumulated research procedures in a wider campaign of hypothesis generation, simulation, and live facility monitoring under conservation laws and digital

twins [201].

A successful setup can become a reusable experimental procedure; diagnosing a failed simulator run can preserve a troubleshooting method; an unproductive search can identify a route that later agents need not repeat. Such knowledge depends on the simulator version, experimental assumptions, and status of the supporting observation. A provisional diagnosis and a replicated finding should retain those distinct statuses when incorporated into campaign memory, so that later work can reuse the procedure without silently strengthening the evidence behind it.

Independent assessment is particularly important when the harness also revises its internal checks. Autonomous research agents have declared success on failed implementations and evaluations [202], and frontier-model case studies in mathematics and science record confident errors caught by experts [203]. A flawed verifier or a loosened test can let later runs build on internally consistent but invalid findings [194]. Final assessment must remain outside the mutable research procedure, with evidence appropriate to the object being studied: execution-based correctness tests, held-out benchmarks, or physical experiments [204, 205]. The cost and delay of these checks affect how quickly a campaign can incorporate new experience, a capacity constraint taken up in Section 8.3.

7.4 Robotics and Embodied Interaction

Embodied harness updates must survive changes in objects, hardware, and contact conditions. With a fixed perception–action backbone, the mutable state includes executable skills, API wrappers, spatial memory, and recovery procedures [12]. EMHO revises monitoring, visual-tool use, grounding, and recovery from execution traces [206]. RoboRSI maintains a task–skill tree around an API-accessed vision-language model and compiles successful manipulation runs into code that must pass a no-regression check in simulation before admission [207].

The transfer from simulation to hardware is a separate test of the learned procedure. Memory Transfer Planning adapts code-based skills to new robot hardware [208], and LYRA stores human-approved manipulation skills [209]. MineEvolve studies reusable skills and failure remedies in an open-world game [210]. Weight-learning systems use experience differently: RoboCat retrains a generalist model [211], and EmbodiedSWE turns verified simulation solutions into robot-policy training trajectories [212]. Their persistent adaptation is in policy weights rather than the harness state considered here.

A skill that passes in simulation still needs staged hardware trials. Reverting a skill library cannot restore a collided part or a damaged workspace; property loss and physical harm are explicit failure categories in agent safety evaluations [213]. Simulation checks such as RoboRSI’s [207] can precede deployment, and action specifications can enforce constraints before a tool executes [214]. Memory transfer has been demonstrated on hardware [208], but discrepancies between simulated and physical conditions require emergency stops and human overrides outside the loop’s edit surface.

7.5 Public Services and Clinical Systems

Clinical and public-service workflows offer recurring opportunities to refine triage, eligibility routing, and case processing. Harness revisions expose these changes in inspectable instructions and procedures around a fixed model, providing an alternative to frequent fine-tuning and its training costs and potential forgetting [215]. The source of a revision matters: implementing a new professional guideline changes the governing rule, whereas adapting execution under an existing guideline changes how that rule is applied.

Current evidence comes mainly from benchmarks and prototypes. LegalBench and GDPval evaluate legal reasoning and professional tasks [216, 217]; MedAgentBench supplies an interactive FHIR environment [218], AgentClinic simulates clinical dialogues from EHR-derived cases [219], and LiveMedBench supplies time-stamped medical evaluation streams [179]. MCE adapts external context and skills on symptom-diagnosis and legal benchmarks [51]. These studies provide settings

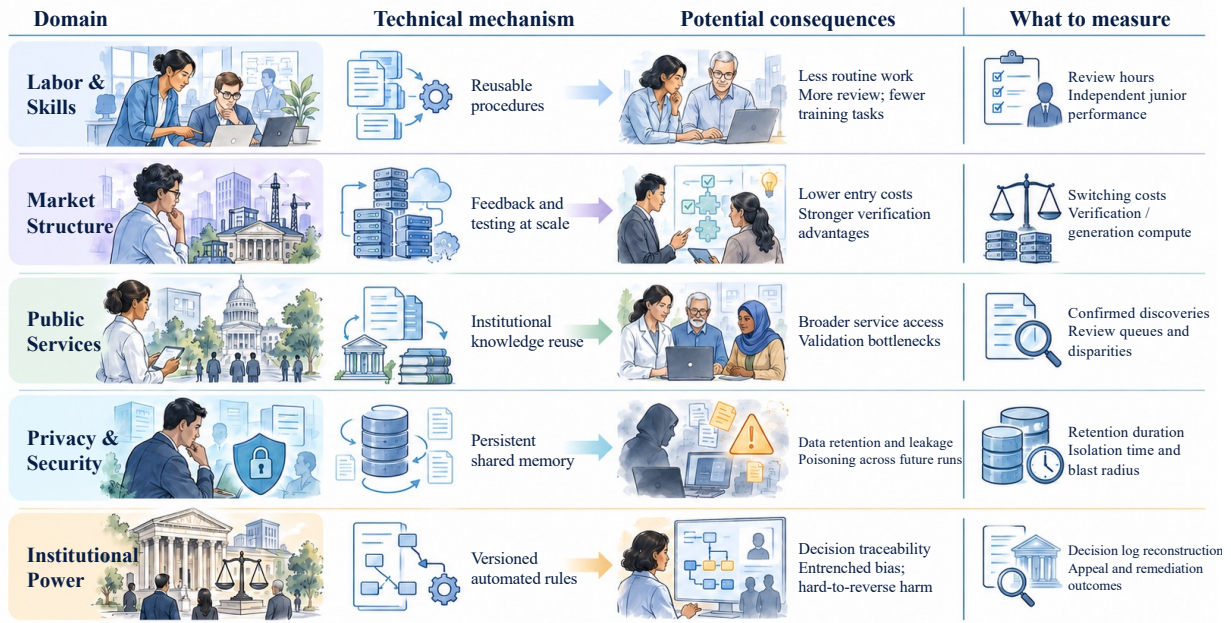


Figure 8 Societal and economic implications of harness self-evolution. The five rows connect reusable procedures, feedback and testing at scale, institutional knowledge reuse, persistent shared memory, and versioned automated rules to potential effects on labor and skills, market structure, public services, privacy and security, and institutional power. The rightmost column lists indicators for empirical assessment, including review effort, switching costs, verification capacity, data retention, and review and remediation outcomes. Arrows denote hypothesized pathways rather than established causal estimates; the balance of benefits and harms depends on deployment conditions and governance.

for assessing candidate procedures.

MedRSI makes the update loop explicit in medical diagnosis [220]. It holds the reasoning model and general workflow fixed while expanding a registry of tool descriptions, executable functions, and specialist model checkpoints. Failures are prioritized by clinical consequence; candidate tools are frozen and tested on subsequent patient cohorts before registration, with reevaluation if the surrounding registry changes. Its *fast discovery, slow registration* mechanism separates creating a capability from admitting it into later diagnostic use. Experiments on public benchmarks and retrospective private cohorts illustrate this mechanism, rather than deployment in clinical care.

Professional oversight must connect operational feedback to the rule a revision is authorized to implement. A candidate derived from failures should be tested on cases the loop did not see and admitted by qualified professionals; local operational success cannot authorize a policy change. Oversight must also follow accumulated state: poisoned memory entries [73] and declining safety behavior as memory grows [221] can shift the system’s behavior without one conspicuous faulty revision. Recording each rule version and the basis for its approval connects that history to later clinical or administrative actions. Section 8.5 considers the consequences for affected people and their ability to contest those actions.

8 Societal and Economic Implications

Persistent harness state makes execution experience available for reuse beyond the person, task, or organization that produced it. Control over those artifacts affects who benefits from accumulated knowledge and who is exposed to its errors. Figure 8 connects these mechanisms to potential effects on work, markets, public services, security, and institutional authority. Longitudinal evidence specific to harness self-evolution is scarce; the discussion distinguishes findings from adjacent settings from hypotheses about evolving systems.

8.1 Productivity, Work, and Education

A reusable procedure can preserve the result of diagnosing unfamiliar dependencies, tuning API parameters, or finding a valid control flow [90, 117]. Later tasks can use that knowledge without repeating the same discovery process. The resulting savings depend on how often the procedure applies and how much work its execution and maintenance still require.

Organization-level productivity depends on how those saved steps affect the rest of the workflow. In a study of GitHub Copilot adoption against historical engineering metrics, per-developer efficiency gains did not appear as measurable organizational throughput [222]. Task-automation accounts and per-task Copilot studies provide further evidence from adjacent settings [217, 223–225], but do not measure harness self-evolution. Reusable procedures could shift work from routine synthesis toward problem formulation and failure verification; whether that shift improves throughput depends on the time and expertise those remaining activities require. Review hours and task lead times [226] can reveal whether saved agent work also reduces the burden on the surrounding organization.

Reusing diagnostic procedures may also change how professionals learn to diagnose. Junior staff acquire judgement through debugging, drafting, and failure triage, precisely the activities a harness can turn into reusable artifacts. A randomized experiment with 52 developers learning an unfamiliar library found significantly lower subsequent quiz scores with AI assistance and no significant average speed-up. Within the assisted group, cognitively engaged interaction patterns were associated with better learning; this subgroup result was observational [227]. The concern for an evolving harness is whether saved procedures remove learning opportunities or expose failures in a form that helps staff develop the judgement needed to review later updates.

Fixed-configuration tutoring studies show how assistance can affect later independent performance. In a field experiment with nearly a thousand high-school mathematics students, relative to the no-AI control, a standard GPT-4 interface raised grades during assisted practice by 48% and lowered them by 17% once access was removed, whereas a tutor built on the same model with prompts designed to safeguard learning raised assisted grades by 127% and largely removed the later deficit [228]. The two arms differ in configuration rather than in model, which places the effect at the layer a harness update edits. Laboratory and field studies likewise find that using an LLM to substitute for learning activities reduces understanding while complementary use increases it [229], and a tutor with structured prompts and scaffolding outperformed in-class active learning in an undergraduate physics course [230].

For a self-evolving tutor, the objective must include what learners can do after assistance is removed. Optimizing assisted performance alone would miss the loss observed in the unguarded configuration above. Research tutors already accumulate cross-session learner memory or optimize tutoring prompts with fixed model weights, but the systems we located evaluate updates against simulated students [231–233]. Their effects on real students’ unassisted learning, and any population-scale consequences [234], remain unestablished. Measuring independent performance after the tool is removed would test whether evolving assistance helps learners acquire capabilities they can retain.

8.2 Economic Distribution and Market Structure

Who gains from these savings depends first on who controls the accumulated knowledge. Employees and users produce the traces a harness learns from, while an operator may control access to and distribution of the resulting playbooks and skills; legal ownership depends on the applicable contracts and rules. Pooling many users’ trajectories, as in SkillClaw [37], can concentrate the value of that reusable experience at the platform.

Accumulated state can also raise switching costs. An organization may have invested in procedures, memories, and tuned workflows that it would need to rebuild after moving to another

system. Portable formats such as file-based skills [65] preserve part of this investment. The destination must still supply compatible tools, relevant private context, and checks appropriate to its environment. Open artifact formats therefore make transfer possible while leaving adaptation and validation costs that can sustain platform dependence.

Verification capacity may shape competition as much as access to a base model. Specialized firms can adapt a fixed model’s harness without financing pre-training, but checking a proposed change for silent regressions can require a test suite, container farm, or physical trial [201, 204, 205]. Incumbents with continuous user traffic collect more failure traces and can afford more checking. This supports broader searches and more extensive checks for regressions or reward gaming [194, 195], giving testing capacity a potential role in concentrating the gains from accumulated experience.

Shared verification infrastructure can lower part of that barrier. Execution-based benchmarks release repository tests, GPU-kernel checks, and machine-learning graders [5, 205, 235]; automated collection pipelines refresh task suites [143, 144]; harness-optimization benchmarks provide held-out partitions and resource accounting [131, 132]. Operators still pay to run and maintain these systems and obtain private test data. Whether portable artifacts and shared verifiers offset the advantages of accumulated experience and testing capacity remains an empirical question, with switching costs and performance gaps between enterprise and open systems providing observable consequences.

8.3 Innovation and Public-Service Capacity

Persistent harness state changes how institutions keep knowledge. In long scientific and engineering projects, progress often stalls when key personnel leave or unwritten lab conventions fade; experimental parameters, tool tricks, and error-recovery paths kept in durable, versioned repositories can preserve continuity across long-running campaigns [117, 198, 199]. Knowledge kept in artifacts is not necessarily kept by staff, however, which ties institutional continuity to the skill-formation concern of Section 8.1. In public administration, healthcare intake, legal aid, and public education, the same persistence could let resource-constrained agencies offer procedural assistance at lower marginal cost. Benchmarks for clinical and legal agent tasks exist [216, 218, 219], but effects on waiting queues and service reach have not been measured, and the evidence on learning is discussed in Section 8.1.

Reusable procedures can reduce setup work, guide triage, and help select which proposals receive scarce expert or experimental attention. Their contribution to capacity depends on what remains after these savings. Generating a scientific hypothesis or a clinical plan can be fast, while the appropriate confirmation may require a physical experiment or professional review [201]. Such capacity does not automatically grow with proposal volume. EmbodiedSWE illustrates an adjacent constraint in robotics: many verified simulation trajectories feed a limited channel for real-robot execution [212]. Accumulated procedures could improve how this channel is used, but increased generation alone need not increase the number of validated results or completed services.

Errors in retained procedures can instead consume the same scarce resources repeatedly. An incorrect troubleshooting rule in shared memory may lead later campaigns to spend instrument time, reagents, or expert review before its common source is identified. Learning from a loose proxy can similarly preserve shortcut rules whose effects become harder to detect under strong optimization [221, 236]. In public services, repeated administrative errors can add review work and delay access. The institutional benefit of accumulation therefore depends on whether retained knowledge reduces repeated effort and improves completed work, as reflected in validation costs, waiting times, service reach, and error rates.

8.4 Privacy, Security, and Systemic Stability

Execution traces, tool telemetry, and user inputs can carry sensitive information into persistent harness state. API credentials, business logic, or personal details retained in playbooks and memory indexes may outlive the session that supplied them. Memory-injection studies establish a persistence mechanism through which stored content affects later sessions [72, 73]. In pooled libraries such as SkillClaw’s [37], guidance derived from one user’s work may also be retrieved for others who lack access to the original data.

Deleting the source records may leave derived skills or memory entries unchanged. An artifact can continue to influence behavior even when it no longer contains an identifiable copy of its source. Addressing a deletion request therefore requires provenance connecting inputs to derived artifacts, so that the retained influence can be located and removed or revised.

Persistence also extends the life of an attack, and checking that a learned skill succeeds on tasks does not reliably stop it. Malicious interaction trajectories can induce an agent to generate skills that are then persistently stored and repeatedly activated [193], and payloads planted in memory files attack current and future sessions [72]. Poisoning does not even require a malicious trajectory: successful experiences that are each task-correct and pass verification can be assembled so that the distilled skill applies a target behavior outside the conditions that constrained it, with an attack success rate of 95.71% in the reported setting [237]. In self-evolving coding agents that author new tools by imitating skills retrieved from a shared library, a malicious skill can become the template for new skills that keep its payload, and the copies re-enter the library and persist after the planted skills are removed [165].

Persistent revisions can also introduce safety failures without adversarial input. SEABench pairs evolving agents with non-evolving ones on the same 48 task sequences. The evolving agents completed more tasks but also showed safety failures absent from the paired baselines, with the divergence visible in their own reasoning [238].

Sharing introduces a further source of correlated failure. Analyses of algorithmic monoculture show how decision-makers using a common component can share errors and impose the same adverse outcomes on the same individuals [239, 240]. Portable harness artifacts provide a route for such dependence beyond the base model: skills, tool servers, and memory files can be copied, combined, and regenerated across deployments. Public registries such as ClawHub and the Hermes Skills Hub distribute these artifacts [186, 187]. Measurements find vulnerability patterns in marketplace skills, concentrated malicious supply, malicious tool servers passing platform audits, and disagreement among security scanners [241–244]. Shared dependencies in the Model Context Protocol ecosystem [245], cross-user retrieval of injected memory [246], and prompts that replicate across applications [247] identify mechanisms through which exposure can spread.

These mechanisms suggest that a common faulty artifact can correlate failures across deployments, especially when many depend on the same sources and checks. The cited studies establish parts of this pathway rather than its organization-wide consequences. Tracking which deployments load an artifact version and how long its derivatives remain active would make the scale and duration of such exposure measurable.

8.5 Governance and Institutional Power

Persistent harness updates can shift who exercises institutional discretion. Human case workers interpret ambiguous regulations with situational context; an evolving harness records operational choices in retrieval rankings, eligibility filters, and routing policies [248]. When these choices become reusable rules, authority is shared among those who specify the objective, supply the experience, approve revisions, and decide where they are deployed. Organizational bias in execution data can be reproduced in these artifacts, making access to their creation and approval processes consequential for affected people.

Versioned code, prompts, and configuration make this changing rule system inspectable. Connecting a rule to an individual decision also requires records of the loaded state, execution traces, inputs and outputs, the evaluation that admitted the revision, and its approval [153]. These records identify what operated; the comparisons in Section 6.2.3 address its causal contribution. Inspectability supports accountability when affected individuals, representatives, and auditors can access the relevant record and use it to contest a decision or seek revision of a rule.

Contestation must lead to an available remedy. Runtime rollback covers in-process state, while external effects require separate action [115]. Version histories can help locate an improper transfer or medical-record change, and write barriers and staging can prevent some actions before release. After a faulty rule has been used, institutions must decide which cases to reopen, who can authorize correction, and who bears its cost. Reinstating a software version cannot restore a missed treatment opportunity or by itself reverse a denied entitlement. Sharing learned procedures therefore creates responsibilities for both subsequent deployment and decisions already made under a withdrawn rule.

9 Future Directions

Harness self-evolution makes experience available as revisable instructions, procedures, and executable components. The remaining challenge is to make these revisions accumulate into useful capabilities as models, environments, and deployment conditions change. This requires decisions about where to invest adaptation effort, which learned artifacts to retain, and when a change invalidates the evidence or authority under which a system operates.

9.1 Model–Harness–Environment Co-Evolution

Model and harness updates. A failure observed in deployment may call for a harness revision, weight training, or a combination of the two. Systems already evolve both components [249], but the choice depends on what each can change. On one long-horizon planning benchmark, harness revisions addressed blocked calls, loops, and exhausted step budgets, while weight training addressed poor delivered plans; adapters trained on evolved-harness trajectories also carried the harness gain into weights [96]. On a second benchmark in the same study, the gain remained in what the model saw, and training on those trajectories added no further benefit. These cases motivate identifying which failures arise from the execution procedure and which persist after the procedure is revised.

Allocating adaptation effort also requires accounting for subsequent use. A reusable harness procedure incurs context or tool-execution costs, while training incurs an upfront cost that may be recovered over many later tasks. Comparisons of harness-only adaptation, weight-only adaptation, and joint adaptation should include an unchanged baseline on the same failure families, with both adaptation and execution costs reported. Update order matters as well: changing the harness changes the trajectories available for training. CoTrace matches training trajectories to their generating runtime while alternating harness search and policy training [250]; traces produced under a fixed model can also supply later training data [82]. The scheduling problem is to exploit these interactions while preserving capabilities acquired before either component changes.

Designing experience. Environment design determines which experience is available to these update processes. POET jointly generates obstacle-course environments and optimizes their controllers, with transfer between environments providing additional routes to learning [251]. For LLM agents, EnvHarness wraps practice environments to vary initial states and interaction constraints, then extracts skills from the resulting trajectories while retaining the underlying task verifiers [252]. The environment interventions shape learning conditions; the extracted skills become part of the agent’s retained harness. The further question is which practice conditions

broaden a procedure’s range of use, expose an assumption that no longer holds, or supply experience better suited to weight training. Selecting environments by their contribution to later capability would connect curriculum design to the allocation of learning between the model and the harness.

Controlled practice and deployment provide different opportunities for this loop. Simulators and resettable software environments allow repeated trials under chosen conditions; live digital services and physical systems may offer limited opportunities to reproduce an earlier state. Memory Transfer Planning adapts entries from a fixed code memory across simulated environments and to a physical robot [208], illustrating how execution experience can cross this boundary. A continuing loop could use deployment failures to revise practice scenarios, then retain procedures together with the environmental assumptions under which they apply. As interfaces, operating rules, or physical conditions change, the challenge is to preserve reusable structure while revising those assumptions. This loop would couple deliberate practice design to adaptation in a changing deployment environment; Section 9.5 considers how to assess accumulated capability under these changes.

Portability across models. Model replacement creates another change in the conditions under which accumulated harness state must remain useful. Across 66 model–harness configurations, model rankings reverse between harnesses, and vendor pairings are not reliably best [253]; searched harnesses have also transferred to unseen models [101]. Pairing sensitivity makes later model upgrades a test of which parts of accumulated harness state remain useful. On each later checkpoint, comparing a frozen evolved harness with both an unevolved baseline and a harness re-evolved for that checkpoint would distinguish retained gains from gains requiring further adaptation. Using the same successor checkpoint in all three arms isolates these differences from the effect of replacing the model.

9.2 Formal Representations for Harness Evolution

Semantics and contracts. Harness evolution raises a programming-language question: how can experience accumulate as composable programs with explicit assumptions about their use? LMQL formalizes parts of model interaction and constrained decoding [254], while Cordis develops a calculus for dynamic component composition [115]. JAZ treats model invocation as a language primitive: `invoke` generates executable implementations at call time, permits recursive invocations, and exposes inputs and interaction history as variables in the code environment [255]. Its hooks support monitoring, return validation, and resource limits. These approaches expose different parts of a harness to programmatic control. Connecting them requires semantics for context, memory, persistent state, and tool effects, with explicit assumptions about model outputs and external systems.

Lean 4 represents definitions and proofs within a common logical foundation [256]; LeanDojo exposes proof states, tactic execution, and library premises to external search [257]. An analogous harness interface could specify states, permitted transitions, and contracts independently of the policy that proposes a step. A model call could remain nondeterministic while its output must pass a declared check before changing persistent state. For example, a revised completion procedure could be required to return the same artifact version that passed validation. The research problem is to define such contracts at a useful granularity and establish their correspondence with actual execution, including the meaning assigned to tool results and stored experience.

Composition and revision. Formalization also changes what can be retained for later tasks. LEGO-Prover stores Isabelle-verified lemmas and their proofs, then generates and verifies more reusable lemmas [258]. VASO uses model-checking counterexamples to revise reusable skill contracts [60]. These mechanisms connect experience to reusable structures and evidence about their properties. Extending them to interacting harness components requires reliable semantic

abstractions: VASO identifies its unverified state-to-proposition mapping as a limitation, and its checks assume sequential skill execution. A compositional account must specify how one component’s guarantees satisfy another’s assumptions, including when shared state or concurrent execution changes those relationships.

Continual revision raises a further distinction. Semantics-preserving transformations can optimize execution, whereas learning intentionally changes behavior and should preserve selected contracts rather than require equivalence with the previous program. Candidate updates could carry proof obligations for the properties they retain, with changed assumptions triggering new obligations in dependent components. Proof assistants, SMT solvers, or model checkers could discharge different parts of these obligations while the trusted checking rules remain outside the editable search process. The open problem is to reuse established evidence across revisions without overlooking changed dependencies. A lemma may remain formally valid while changes to retrieval or search make it harder to use. The value of formal representations for cumulative capability must therefore be assessed through later-task performance together with the cost of acquiring, checking, and reusing the retained structures.

9.3 Synthesis and Reuse of Harness Plugins

Plugin creation can turn a repeatedly needed operation into a capability that later tasks can invoke. Creating one requires choosing which parts of a successful execution to preserve, which inputs to expose, and which task-specific assumptions to leave outside its interface. A specialized implementation may serve a narrow workload efficiently; a more general one may support wider reuse but require more complex checks. The open problem is to infer a useful scope from limited experience and to revise that scope as new tasks reveal where the capability applies.

Whether creation pays off depends on the plugin’s subsequent lifecycle. Generation and validation incur upfront costs, followed by retrieval, execution, distribution, and maintenance. A frequently used capability can recover its creation cost, while a rarely reused one may become obsolete after a workload or interface change. Comparing reuse with further revision against regeneration on the same task sequence, at matched task quality, would identify when retaining a plugin is worthwhile. Failed generations, unsuccessful reuse, and maintenance after dependency changes belong in this accounting. Related procedures may also be consolidated, while unused or superseded versions can leave the active library.

The contracts discussed in Section 9.2 can delimit part of the evidence reused with a plugin. Empirical validation also depends on the checked composition, model, dependencies, and task conditions; changing one can affect the plugin even when its source stays fixed. Dependency-aware revalidation could preserve unaffected checks and repeat those whose assumptions changed. Comparing this policy with full revalidation under the same changes would measure both the saved work and any regressions it misses.

9.4 Distributed Execution and Edge Infrastructure

Harness evolution makes the locations of capability generation, execution, and validation separate design choices. A generated tool can run near private data, reducing repeated data transfers, while centralized execution can simplify shared updates. A further arrangement keeps cached skills and interaction on the client while centralizing synthesis and regression tests. These choices depend on network conditions, device resources, data residency, and how frequently the workload reuses an artifact. Placement should therefore be assessed over a sequence of updates and uses, including synthesis, validation, distribution, and execution costs at comparable task quality.

Distribution also connects deployment consistency to local compatibility. A client must support a tool’s dependencies, and verification must cover the environment in which it will execute. MOSS and Cordis provide mechanisms for deployment and recovery [108, 115]; shared libraries such as SkillClaw’s add revisions contributed by multiple users [37]. Concurrent updates can produce

combinations that were never checked together. Promotion and execution records therefore need to identify the component versions actually used, either as a stable snapshot or as a sequence of version changes within a run. Hot reloading [115] makes this a concern even within one continuing session.

Disconnected clients add a choice between continued availability and timely withdrawal. A cached capability may remain useful while the network is unavailable, yet the server may have replaced or withdrawn it in the meantime. Update frequency, the consequences of stale behavior, and the allowed lifetime of a cached version should determine when local execution can continue and when it must await synchronization.

9.5 Dynamic Evaluation and Benchmark Refresh

Adaptation and retention. Changing environments alter what retained capability should do. ServeLearnBench makes hidden operating rules change, disappear, and recur across a task stream [160]. A useful harness must stop applying obsolete rules while preserving procedures whose assumptions still hold. Longitudinal evaluation therefore needs to distinguish learning a new condition, retaining competence under unchanged conditions, and reusing earlier knowledge when a condition returns. Frozen transfer and adaptation with additional experience answer different questions and should be reported separately. This distinction also matters for the environment-design loop in Section 9.1: success in a customized practice environment does not establish that its lessons remain useful after the scaffolding or constraints change. Evaluations should follow the resulting harness state onto separate deployment tasks, accounting for the experience and resources used in further adaptation.

Reference design. Where earlier execution conditions can be reconstructed, the crossed design in Section 6.3.3 can compare frozen harness states across task and environment snapshots, including disjoint task sets. The research problem is how to maintain informative references within a limited budget, and what can still be established when a historical service or physical state cannot be replayed. Anchor tasks can reduce repeated cross-evaluation costs and preserve measurements on a common subset, alongside comparisons on refreshed tasks. Resources must be allocated among retained anchors, reevaluation of earlier harness states, and fresh tasks; the choice determines which changes and regressions remain detectable. Controlled replicas can preserve selected conditions, but their fidelity limits what a replay establishes about the original deployment.

Different benchmark designs provide different parts of this reference. Time-stamped tasks in LiveBench, LiveCodeBench, and LiveMedBench limit direct overlap with earlier development [177–179]; version-pinned environments in Terminal-Bench help reproduce execution conditions [259]; controlled transformations such as those in StressEval expose sensitivity to task factors [176]. These controls support comparability. Independence additionally depends on how their outcomes are used. Once anchor results guide a revision or select a checkpoint, they become search feedback [136, 260]. Subsequent claims then need separate reporting evidence or inference that accounts for adaptive reuse. Replacing anchors without recording their prior exposure does not resolve this distinction.

Evaluator calibration. A changed operating rule may make a formerly correct action wrong even under a consistent scoring procedure. Changes in the scoring procedure itself require a separate reference. LLM judges can overestimate agent success and reverse expert rankings [150], and a judge used in iterative self-improvement has drifted toward maximal scores and longer outputs [261]. Rescoring a shared sample of outcomes with both evaluator versions under the same task specification, alongside executable checks or expert judgments, would help distinguish a measurement change from a harness change. This calibration consumes resources that are also needed for admission and regression testing. Budget allocation should reflect the consequences of

undetected errors, including specification gaming [151] and safety regressions [221], rather than only the precision of an average task score.

9.6 Safety Boundaries for Self-Modifying Systems

Creating a new capability must not automatically grant authority to use it. For a continually revised harness, authorization must remain tied to the operations, data, and deployment conditions covered by a grant. A tool approved to read internal records may later acquire an export step, while combining permitted tools can create a data flow outside the original authorization. Specifications over data flow and tool sequences offer a basis for enforcing such limits [214]. The contracts discussed in Section 9.2 could connect an authorization to explicit behavioral assumptions and identify revisions that require renewed authorization. The mechanism enforcing these limits must remain outside the editable loop, including limits on physical actions or spending whose effects cannot be reversed by restoring software state [115].

Safety during learning also depends on which behavior becomes reusable. SEABench reports safety failures arising from persistent updates without direct adversarial influence [238]. SkillPoison deliberately induces harmful generalization from individually verified successful experiences by removing conditions that constrain where the learned behavior applies [237]. These findings motivate learning the conditions under which a behavior is appropriate and retaining them alongside the procedure. Tests that vary those conditions can probe whether the procedure still acts appropriately; constraints that can be expressed over execution can also be enforced when it runs. The challenge is to broaden a procedure’s use while detecting when new applications invalidate its safety assumptions.

Reuse can also propagate malicious behavior across generations. EVOMAL shows how newly authored skills inherit malicious code from retrieved examples and remain in circulation after the planted skills are removed. It studies a counter-prompt against this copying and proposes a signed quarantine gate that keeps newly authored skills outside the retrievable library [165]. Extending such controls to continued learning requires deciding which artifacts can enter reuse and under what evidence. The cost of an admission policy includes useful capabilities whose reuse it delays or prevents. Longitudinal comparisons should track that loss alongside the continued activation of harmful derivatives and the time needed to contain their propagation.

9.7 Governance, Provenance, and Human Oversight

Human-in-the-loop oversight. The update policy must specify which changes require human judgement and what that intervention can alter. A user or institution can delegate classes of updates subject to declared conditions, while reserving changes in objectives, affected users, or data access for renewed authorization. The unit of oversight is therefore a design choice: an individual revision, a capability and its conditions of use, or a release shared across deployments. Each level needs a responsible party with authority to approve, restrict, or withdraw use. The challenge is to maintain this allocation of authority as locally acquired procedures spread across users and organizations.

Effective oversight depends on what a reviewer can infer about a change. A short memory edit may redirect many later tasks, and an unchanged tool may behave differently when composed with a new skill. Review evidence should connect proposed edits to changes in exercised behavior, assumptions, and affected deployments. Grouping revisions into reviews of a capability could reduce repeated work, but the summary must expose changes that alter the authorized scope. Research on review interfaces should examine whether people detect such changes and choose effective interventions, alongside the effort and delays those decisions require.

Provenance and withdrawal. Provenance can supply part of this evidence. HAL records model and tool calls for inspection [153], while ScientistOne connects research claims to supporting

artifacts [262]. Extending these mechanisms to harness evolution would link source experience, a proposed revision, its admission decision, and the state actually loaded in subsequent runs. A generated skill may combine or reinterpret several sources, so derivation cannot always be recovered from file history or textual similarity. Recording the inputs supplied during generation establishes exposure; determining their contribution to later behavior requires further evidence. The research problem is to preserve useful lineage through synthesis and reuse while making gaps in that lineage explicit.

Withdrawal adds an operational requirement to this history. An operator needs to know where affected state remains active, which derivatives require inspection, and who can suspend use in each deployment. Reconstructing a disputed decision requires the state and evidence present at that time; acting on a withdrawal requires a current account of use. Incomplete lineage creates a choice between leaving potentially affected derivatives active and withdrawing a wider set at greater disruption cost. Records of past use should also help institutions locate earlier decisions that may require reconsideration under the correction and contestation procedures of Section 8.5.

10 Conclusion

Harness self-evolution gives an agent a way to revise its working procedures from execution experience while keeping its base model fixed. Instructions, memories, workflows, and executable components provide different places for that experience to accumulate. Their value lies in how they change later execution: a lesson must become usable guidance or code, reach the tasks where it applies, and remain useful as the surrounding system changes. The update targets reviewed here expose different parts of this process to search and validation.

Evaluation must follow these changes through time. A held-out comparison of the final system, a trace showing that a patch ran, and a check used to admit that patch support different conclusions. Selection, activation, attribution, and improvement locate the evidence needed for each of these conclusions. Retention also concerns more than preserving an artifact: an unchanged skill can lose influence, while a merged or revised procedure may preserve the capability it supported. Later-task probes and controlled comparisons are needed to establish what survives. The theoretical analysis complements these protocols by relating candidate search to evolutionary mechanisms and giving sufficient conditions for confidence-adjusted admission under explicit sampling and error-control assumptions.

Useful accumulation depends on selective maintenance. Reuse can save repeated discovery, but verification, execution, and adaptation to new conditions still incur costs. Shared procedures can carry both knowledge and errors across users and deployments, making their conditions of use, provenance, and withdrawal part of the learning system. Progress in harness self-evolution should therefore be assessed by the capabilities that agents can continue to exercise as experience accumulates, together with the resources and oversight needed to sustain them.

References

- [1] Irving John Good. Speculations concerning the first ultraintelligent machine. In Franz L. Alt and Morris Rubinfeld, editors, *Advances in Computers*, volume 6, pages 31–88. Elsevier, 1965.
- [2] Jürgen Schmidhuber. Gödel machines: Self-referential universal problem solvers making provably optimal self-improvements, 2003. arXiv:cs/0309048.
- [3] Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Qihan Ren, Yiran Wu, Hongru Wang, Han Xiao, Yuhang Zhou, Shaokun Zhang, Jiayi Zhang, Jinyu Xiang, Yixiong Fang, Qiwen Zhao, Dongrui Liu, Cheng Qian, Zhenhailong Wang, Minda Hu, Huazheng Wang, Qingyun Wu, Heng Ji, and Mengdi Wang. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *Transactions on Machine Learning Research*, 2026. URL <https://arxiv.org/abs/2507.21046>.
- [4] Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang, Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu, Siwei Liu, Zihao Li, Zhaochun Ren, Nikos Aletras, Xi Wang, Han Zhou, and Zaiqiao Meng. A comprehensive survey of self-evolving AI agents: A new paradigm bridging foundation models and lifelong agentic systems, 2025. URL <https://arxiv.org/abs/2508.07407>.
- [5] Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Madry. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *International Conference on Learning Representations*, 2025.
- [6] 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, pages 19632–19642. AAAI Press, 2024. doi: 10.1609/aaai.v38i17.29936.
- [7] 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*, 2024, 2024. URL <https://openreview.net/forum?id=ehfRiFOR3a>.
- [8] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.
- [9] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*, Singapore, 2025. URL <https://openreview.net/forum?id=z5uVAKwmjf>.
- [10] Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent, 2025. URL <https://arxiv.org/abs/2504.15228>. arXiv preprint.
- [11] Chenyu Zhou, Huacan Chai, Wenteng Chen, Zihan Guo, Rong Shan, Yuanyi Song, Tianyi Xu, Yingxuan Yang, Aofan Yu, Weiming Zhang, Congming Zheng, Jiachen Zhu, Zeyu Zheng,

- Zhuosheng Zhang, Xingyu Lou, Changwang Zhang, Zhihui Fu, Jun Wang, Weiwen Liu, Jianghao Lin, and Weinan Zhang. Externalization in LLM agents: A unified review of memory, skills, protocols and harness engineering, 2026. URL <https://arxiv.org/abs/2604.08224>.
- [12] Xuying Ning, Katherine Tieu, Dongqi Fu, Tianxin Wei, Zihao Li, Yuanchen Bei, Jiaru Zou, Mengting Ai, Zhining Liu, Ting-Wei Li, Lingjie Chen, Yanjun Zhao, Ke Yang, Bingxuan Li, Cheng Qian, Gaotang Li, Xiao Lin, Zhichen Zeng, Ruizhong Qiu, Sirui Chen, Yifan Sun, Xiyuan Yang, Ruida Wang, Rui Pan, Chenyuan Yang, Dylan Zhang, Liri Fang, Zikun Cui, Yang Cao, Pan Chen, Dorothy Sun, Ren Chen, Mahesh Srinivasan, Nipun Mathur, Yinglong Xia, Hong Li, Hong Yan, Pan Lu, Lingming Zhang, Tong Zhang, Hanghang Tong, and Jingrui He. Code as agent harness, 2026. URL <https://arxiv.org/abs/2605.18747>. arXiv preprint.
- [13] Zhe Ren, Yimeng Chen, Dandan Guo, Guowei Rong, Tonghui Li, R. B. Xiong, Qingfeng Lan, Wenyi Wang, Li Nanbo, Yibo Yang, Mingchen Zhuge, and Jürgen Schmidhuber. Self-improvements in modern agentic systems: A survey, 2026. URL <https://arxiv.org/abs/2607.13104>.
- [14] Mingguang Chen, Licheng Wang, and Bo Qu. Recursive self-improvement in AI: From bounded self-refinement to autonomous research loops, 2026. URL <https://arxiv.org/abs/2607.07663>. survey; arXiv preprint.
- [15] Hao Zhou, Haichuan Hu, Tianyu Luo, Ye Shang, Chunrong Fang, Zhenyu Chen, Liang Xiao, and Quanjun Zhang. Self-evolving coding agents, 2026. arXiv:2608.03392.
- [16] DeepSeek-AI. DeepSeek harness developer preview. <https://www.deepseek.com/harness/en/>, 2026. Product/developer-preview page.
- [17] Douglas B. Lenat. EURISKO: A program that learns new heuristics and domain concepts. *Artificial Intelligence*, 21(1–2):61–98, 1983.
- [18] Chrisantha Fernando, Dylan Sunil Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 13481–13544. PMLR, 2024. URL <https://proceedings.mlr.press/v235/fernando24a.html>.
- [19] Jenny Zhang, Shengran Hu, Cong Lu, Robert Tjarko Lange, and Jeff Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents. In *The Fourteenth International Conference on Learning Representations*, 2026. URL <https://openreview.net/forum?id=pUpzQZTvGY>. Poster.
- [20] Krista Opsahl-Ong, Michael J. Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 9340–9366, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.525. URL <https://aclanthology.org/2024.emnlp-main.525/>.
- [21] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *The Eleventh International Conference on Learning Representations*, 2023. URL <https://openreview.net/forum?id=92gvk82DE->.

- [22] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=Bb4VGOWELI>.
- [23] Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7957–7968, Singapore, 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.494. URL <https://aclanthology.org/2023.emnlp-main.494/>.
- [24] Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative AI by backpropagating language model feedback. *Nature*, 639(8055):609–616, 2025. doi: 10.1038/s41586-025-08661-4.
- [25] Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziemis, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *The Fourteenth International Conference on Learning Representations*, 2026. URL <https://openreview.net/forum?id=RQm2KQTM5r>. Oral.
- [26] Ching-An Cheng, Allen Nie, and Adith Swaminathan. Trace is the next AutoDiff: Generative optimization with rich feedback, execution traces, and LLMs, 2024. URL <https://arxiv.org/abs/2406.16218>. NeurIPS 2024.
- [27] Guowei Xu, Mert Yuksekgonul, Carlos Guestrin, and James Zou. metaTextGrad: Automatically optimizing language model optimizers, 2025. URL <https://arxiv.org/abs/2505.18524>.
- [28] Yang Zhao, Pu Wang, and Hao Frank Yang. How to auto-optimize prompts for domain tasks? adaptive prompting and reasoning through evolutionary domain knowledge adaptation, 2025. URL <https://arxiv.org/abs/2510.21148>.
- [29] Wangcheng Tao, Han Wu, and Weng-Fai Wong. SePO: Self-evolving prompt agent for system prompt optimization, 2026. URL <https://arxiv.org/abs/2606.04465>.
- [30] Tao Tao, Guanghui Zhu, Lang Guo, Hongyi Chen, Chunfeng Yuan, and Yihua Huang. DelvePO: Direction-guided self-evolving framework for flexible prompt optimization, 2025. URL <https://arxiv.org/abs/2510.18257>.
- [31] Anirudh Nair, Adi Banerjee, Laurent Mombaerts, Matthew Hagen, and Tarik Borogovac. Tournament of prompts: Evolving LLM instructions through structured debates and Elo ratings, 2025. URL <https://arxiv.org/abs/2506.00178>.
- [32] Anthropic. Effective context engineering for AI agents. <https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents>, 2025.
- [33] Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, Baolong Bi, Yujun Cai, Jiazhi Liu, Mingyu Li, Zhong-Zhi Li, Duzhen Zhang, Chenlin Zhou, Jiayi Mao, Tianze Xia, Jiafeng Guo, and Shenghua Liu. A survey of context engineering for large language models, 2025. URL <https://arxiv.org/abs/2507.13334>.
- [34] Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Memp: Exploring agent procedural memory. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 17490–17502,

San Diego, California, United States, July 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.findings-acl.866.

- [35] Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 63897–63911. PMLR, 2025.
- [36] Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjin Jiang. Trace2Skill: Distill trajectory-local lessons into transferable agent skills, 2026. arXiv:2603.25158.
- [37] Ziyu Ma, Shidong Yang, Yuxiang Ji, Xucong Wang, Yong Wang, Yiming Hu, Tongwen Huang, and Xiangxiang Chu. SkillClaw: Let skills evolve collectively with agentic evolver, 2026. URL <https://arxiv.org/abs/2604.08377>. arXiv preprint.
- [38] Xiangru Tang, Tianrui Qin, Tianhao Peng, Ziyang Zhou, Daniel Shao, Tingting Du, Xinming Wei, Peng Xia, Fang Wu, He Zhu, Ge Zhang, Jiaheng Liu, Xingyao Wang, Sirui Hong, Chenglin Wu, Hao Cheng, Chi Wang, and Wangchunshu Zhou. Agent KB: Leveraging cross-domain experience for agentic problem solving, 2025. URL <https://arxiv.org/abs/2507.06229>.
- [39] Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling agent self-evolving with reasoning memory. In *The Fourteenth International Conference on Learning Representations*, 2026. URL <https://openreview.net/forum?id=jL7fwchScm>.
- [40] Silin Chen, Shaoxin Lin, Yuling Shi, Heng Lian, Xiaodong Gu, Longfei Yun, Dong Chen, Lin Cao, Jiyang Liu, Nu Xia, and Qianxiang Wang. SWE-Exp: Experience-driven software issue resolution, 2025. URL <https://arxiv.org/abs/2507.23361>.
- [41] Yi-Xuan Deng, Xiaoqin Liu, Yi Zhang, Guo-Wei Yang, and Shuojin Yang. Your code agent can grow alongside you with structured memory, 2026. URL <https://arxiv.org/abs/2603.13258>.
- [42] Haichuan Hu, Guoqing Xie, Quanjin Zhang, Jiawei Liu, Shengcheng Yu, Chunrong Fang, Zhenyu Chen, and Liang Xiao. EvoRepair: Enhancing vulnerability repair agents through experience-based self-evolution, 2026. URL <https://arxiv.org/abs/2605.30105>.
- [43] Kangsan Kim, Minki Kang, Taeil Kim, Yanlai Yang, Mengye Ren, and Sung Ju Hwang. Memory transfer learning: How memories are transferred across domains in coding agents, 2026. URL <https://arxiv.org/abs/2604.14004>.
- [44] Shangyin Tan, Lakshya A. Agrawal, Rohit Sandadi, Dan Klein, Koushik Sen, Alexandros G. Dimakis, and Matei Zaharia. Automatically learning skills for coding agents. In *Proceedings of the ACM Conference on AI and Agentic Systems*, 2026. doi: 10.1145/3786335.3813196. URL <https://doi.org/10.1145/3786335.3813196>.
- [45] Yanzhou Li, Yiran Zhang, Xiaoyu Zhang, Xiaoxia Liu, and Yang Liu. CODESKILL: Learning self-evolving skills for coding agents, 2026. URL <https://arxiv.org/abs/2605.25430>.
- [46] Xingyan Liu, Xiyue Luo, Linyu Li, Ganghong Huang, Jianfeng Liu, and Honglin Qiao. SkillForge: Forging domain-specific, self-evolving agent skills in cloud technical support. In *Proceedings of the 49th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 4763–4768. ACM, 2026. doi: 10.1145/3805712.3808466. Industry Track.

- [47] Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7080–7106, Rabat, Morocco, March 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.eacl-long.333. URL <https://aclanthology.org/2026.eacl-long.333/>.
- [48] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *The Fourteenth International Conference on Learning Representations*, 2026. URL <https://openreview.net/forum?id=eC4ygDs02R>.
- [49] Yifan Yang, Ziyang Gong, WeiQuan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, Kai Qiu, Yuqing Yang, Dongdong Chen, Xue Yang, and Chong Luo. SkillOpt: Executive strategy for self-evolving agent skills, 2026. Preprint, arXiv:2605.23904.
- [50] Yifei Shen, Bo Li, and Xinjie Zhang. SkillOpt-Lite: Better and faster agent self-evolution via one line of vibe, 2026. Preprint, arXiv:2607.03451.
- [51] Haoran Ye, Xuning He, Vincent Arak, Haonan Dong, and Guojie Song. Meta context engineering via agentic skill evolution. In *Proceedings of the 43rd International Conference on Machine Learning (ICML 2026)*, 2026. URL <https://icml.cc/virtual/2026/poster/64296>. arXiv:2601.21557.
- [52] Yuetian Du, Yucheng Wang, He Xu, Jiexu Xu, Shanwen Tan, Bing Zhao, Boyu Yang, Zhijie Xu, Ming Kong, Hu Wei, Jie Liu, and Qiang Zhu. Living-harness is an interactive-agent evolver, 2026. URL <https://arxiv.org/abs/2607.26598>.
- [53] Yifan Wang, Hao Cheng, Xiaomin Li, Yuexing Hao, Hemanth Neelgund Ramesh, Dongwon Jung, Hao Tang, Keru Wang, Chenliang Zhou, Qianhui Wu, Wenlin Yao, Ananth Grama, Andrzej Banburski-Fahey, Baolin Peng, Jaron Lanier, and Jianfeng Gao. Certified long-horizon code agent evolution via validation-gated skill optimization, 2026. URL <https://arxiv.org/abs/2609.32990>.
- [54] Zihan Chen, Songwei Dong, Chengshuai Shi, Peng Wang, Song Wang, Cong Shen, and Jundong Li. The past is prologue: A plug-in controller for selective updates in sequentially evolving LLM memory, 2026. URL <https://arxiv.org/abs/2606.31121>. arXiv preprint.
- [55] Chen Yang, Jiashuo Tian, Ziqi Wang, Xinyin Liu, Meiru Ye, and Junjie Chen. Learning globally reusable skills for coding agents, 2026. URL <https://arxiv.org/abs/2608.06153>.
- [56] Huawei Lin, Peng Li, Jie Song, Fuxin Jiang, and Tieying Zhang. MUSE-Autoskill: Self-evolving agents via skill creation, memory, management, and evaluation, 2026. URL <https://arxiv.org/abs/2605.27366>.
- [57] Jincheng Ren, Siwei Wu, Yizhi Li, Kang Zhu, Shu Xu, Boyu Feng, Ruibin Yuan, Wei Zhang, Riza Batista-Navarro, Jian Yang, and Chenghua Lin. A self-evolving framework for efficient terminal agents via observational context compression, 2026. URL <https://arxiv.org/abs/2604.19572>.
- [58] Jingzhi Gong, Ruizhen Gu, Zhiwei Fei, Yazhuo Cao, Lukas Twist, Alina Geiger, Shuo Han, Dominik Sobania, Federica Sarro, and Jie M. Zhang. SkillMOO: Multi-objective optimization of agent skills for software engineering, 2026. URL <https://arxiv.org/abs/2604.09297>.

- [59] Zimu Wang, Yuling Shi, Mengfan Li, Zijun Liu, Jie M. Zhang, Chengcheng Wan, and Xiaodong Gu. EffiSkill: Agent skill based automated code efficiency optimization, 2026. URL <https://arxiv.org/abs/2603.27850>.
- [60] Yunhao Yang, Neel P. Bhatt, Kevin Wang, Samuel Tetteh, Zhangyang Wang, and Ufuk Topcu. VASO: Formally verifiable self-evolving skills for physical AI agents, 2026. URL <https://arxiv.org/abs/2606.05395>.
- [61] Guibin Zhang, Haotian Ren, Chong Zhan, Zhenhong Zhou, Junhao Wang, He Zhu, Wangchunshu Zhou, and Shuicheng Yan. MemEvolve: Meta-evolution of agent memory systems, 2025. URL <https://arxiv.org/abs/2512.18746>.
- [62] Anthropic. How Claude remembers your project. <https://code.claude.com/docs/en/memory>, 2026. Claude Code documentation.
- [63] Google. Gemini CLI Auto Memory. <https://github.com/google-gemini/gemini-cli/blob/main/docs/cli/auto-memory.md>, 2026.
- [64] GitHub. About GitHub Copilot Memory. <https://docs.github.com/en/copilot/concepts/agents/copilot-memory>, 2026.
- [65] Anthropic. Equipping agents for the real world with agent skills. <https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills>, 2025.
- [66] Aditya Aggarwal and Nahid Farhady Ghalaty. Self-improving AI coding agents through accumulated behavioral rules: A closed-loop framework, 2026. URL <https://arxiv.org/abs/2607.13091>.
- [67] Zisu Huang, Jingwen Xu, Yifan Yang, Ziyang Gong, Qihao Yang, Muzhao Tian, Xiaohua Wang, Changze Lv, Xuemei Gao, Qi Dai, Bei Liu, Kai Qiu, Xue Yang, Dongdong Chen, Xiaoqing Zheng, and Chong Luo. From raw experience to skill consumption: A systematic study of model-generated agent skills, 2026. URL <https://arxiv.org/abs/2605.23899>.
- [68] Zidi Xiong, Yuping Lin, Wenya Xie, Pengfei He, Zirui Liu, Jiliang Tang, Himabindu Lakkaraju, and Zhen Xiang. How memory management impacts LLM agents: An empirical study of experience-following behavior. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2026. doi: 10.18653/v1/2026.acl-long.27. URL <https://aclanthology.org/2026.acl-long.27/>.
- [69] Xijia Tao, Yihua Teng, Xinyu Fu, Zirui Liu, Kecheng Chen, Yuzhi Zhao, Suiyun Zhang, Rui Liu, and Lingpeng Kong. SoftSkill: Behavioral compression for contextual adaptation, 2026. URL <https://arxiv.org/abs/2606.20333>.
- [70] Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, Jiankai Sun, Shuyi Wang, Chujun Tao, Binxu Li, Xuandong Zhao, Hejia Geng, Xiaojun Wu, Junwei Zhou, Xiaokun Chen, Hanwen Xing, Yubo Li, Qunhong Zeng, Di Wang, Yuanli Wang, Roey Ben Chaim, Penghao Jiang, Haotian Shen, Luyang Kong, Xinyi Liu, Runhui Wang, Xuanqing Liu, Jiachen Li, Xin Lan, Yueqian Lin, Wengao Ye, Junwei He, Songlin Li, Yue Zhang, Yipeng Gao, Yijiang Li, Ze Ma, Liqiang Jing, Tianyu Wang, Kaixin Li, Yiqi Xue, Haoran Lyu, Yizhuo He, Yuchen Tian, Shutong Wu, Bowei Wang, Yixuan Gao, Bo Chen, Litong Liu, Sikai Cheng, Jiajun Bao, Shuaicheng Tong, Shuwen Xu, Terry Yue Zhuo, Tinghan Ye, Qi Qi, Miao Li, Longtai Liao, Zelin Tan, Chang Shi, Xilin Tang, Srinath Tankasala, Boqin Yuan, Yaoyao Qian, Jianhong Tu, Chenguang Wang, Yizhou Sun, Wei Wang, Aaron Taylor, Ziyue Yang, Changkun Guan, Zhikang Dong, Xinyu Zhang,

- Steven Dillmann, Han chung Lee, and Dawn Song. SkillsBench: Benchmarking how well agent skills work across diverse tasks, 2026. URL <https://arxiv.org/abs/2602.12670>. arXiv preprint.
- [71] Minhua Lin, Juncheng Wu, Zijun Wang, Zhan Shi, Yisi Sang, Bing He, Zewen Liu, Tianxin Wei, Zongyu Wu, Zhiwei Zhang, Dakuo Wang, Xiang Zhang, Benoit Dumoulin, Cihang Xie, Yuyin Zhou, Suhang Wang, and Hanqing Lu. Harness updating is not harness benefit: Disentangling evolution capabilities in self-evolving LLM agents, 2026. arXiv:2605.30621.
 - [72] Soham Gadgil, David Alexander, Sai Sunku, and Franziska Roesner. Bad memory: Evaluating prompt injection risks from memory in agentic systems, 2026. URL <https://arxiv.org/abs/2607.14611>. arXiv preprint.
 - [73] Pritam Dash, Tongyu Ge, Aditi Jain, Tanmay Shah, and Zhiwei Shang. From untrusted input to trusted memory: A systematic study of memory poisoning attacks in LLM agents, 2026. URL <https://arxiv.org/abs/2606.04329>. arXiv preprint.
 - [74] Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 62743–62767. PMLR, 2024.
 - [75] Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Serkan Ö. Arık. Multi-agent design: Optimizing agents with better prompts and topologies. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/1ab4e0e8f35078109bc78d7b465d306f-Abstract-Conference.html. Originally released as arXiv:2502.02533.
 - [76] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=PFS4ffN9Yx>.
 - [77] Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 75834–75852. PMLR, 2025. URL <https://proceedings.mlr.press/v267/zhang25bi.html>.
 - [78] Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. AgentSquare: Automatic LLM agent search in modular design space. In *The Thirteenth International Conference on Learning Representations*, Singapore, 2025. URL <https://openreview.net/forum?id=mPdmDYIQ7f>.
 - [79] Yixin Ou, Wangchunshu Zhou, Shengwei Ding, Long Li, Jialong Wu, Tiannan Wang, Jiamin Chen, Shuai Wang, Xiaohua Xu, Ningyu Zhang, Huajun Chen, and Yuchen Eleanor Jiang. Symbolic learning enables self-evolving agents. *AI Open*, 6:314–322, 2025. doi: 10.1016/j.aiopen.2025.11.004. Originally released as arXiv:2406.18532.
 - [80] Siwei Liu, Jinyuan Fang, Han Zhou, Yingxu Wang, and Zaiqiao Meng. SEW: Self-evolving agentic workflows for automated code generation, 2025. URL <https://arxiv.org/abs/2505.18646>.

- [81] Zihan Ma, Zhikai Zhao, Chuanbo Hua, Federico Berto, and Jinkyoo Park. JudgeFlow: Agentic workflow optimization via block judge, 2026. URL <https://arxiv.org/abs/2601.07477>.
- [82] Hyunin Lee, Jinglue Xu, Jeffrey Seely, Donghyun Lee, Matei Zaharia, and Yujin Tang. Recursive harness self-improvement, 2026. URL <https://arxiv.org/abs/2607.15524>. arXiv preprint.
- [83] Zhezheng Hao, Tianfu Wang, Huanshuo Dong, Ziyang Liu, Hong Wang, Xiankun Lin, Qiang Lin, Can Wang, Hande Dong, and Jiawei Chen. Evolve as a team: Collaborative self-evolution for LLM-based multi-agent systems, 2026. URL <https://arxiv.org/abs/2605.29790>.
- [84] Simon Yu, Gang Li, Weiyan Shi, and Peng Qi. PolySkill: Learning generalizable skills through polymorphic abstraction, 2025. URL <https://arxiv.org/abs/2510.15863>.
- [85] Zeyu Ren, Ling Yue, Ran Li, Yishu Wang, Shengxiang Xu, Hanmo Liu, Shaowu Pan, and Shimin Di. FlowEvo: Self-evolving agents through the co-evolution of workflows and executable skills, 2026. URL <https://arxiv.org/abs/2607.21596>.
- [86] Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. Live-SWE-agent: Can software engineering agents self-evolve on the fly?, 2025. URL <https://arxiv.org/abs/2511.13646>.
- [87] Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-harness: Harnesses that improve themselves, 2026. Preprint, arXiv:2606.09498.
- [88] Mengzhuo Chen, Junjie Wang, Zhe Liu, Yawen Wang, Haiming Zheng, and Qing Wang. From failed trajectories to reliable LLM agents: Diagnosing and repairing harness flaws, 2026. URL <https://arxiv.org/abs/2606.06324>.
- [89] Luan Zhang, Ruochen Zhou, Dandan Song, Zhengyu Chen, Yuhang Tian, Jun Yang, Huipeng Ma, Chenhao Li, Guangyuan Feng, Xudong Li, Yizhou Jin, and Yan Xu. HarnessCompass: Guiding automatic harness evolution toward generalizable and effective agent harnesses, 2026. URL <https://arxiv.org/abs/2608.01918>.
- [90] Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses, 2026. arXiv:2604.25850.
- [91] Sungho Park, Wonjoong Kim, Rongyuan Tan, Jue Zhang, Wook-Shin Han, Pengfei Gao, Chanyoung Park, Yongqiang Yao, Rao Fu, Elsie Nallipogu, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. AutoSaddler: Automatic harness optimization with durable updates from agent execution traces, 2026. URL <https://arxiv.org/abs/2608.23041>.
- [92] Tianxin Wei, Zhan Shi, Minhua Lin, Bing He, Zewen Liu, Yisi Sang, Yuanchen Bei, Xuying Ning, Jiaru Zou, Ting-Wei Li, Xiao Lin, Yanjun Zhao, Chi Wang, Benoit Dumoulin, Dakuo Wang, Jingrui He, and Hanqing Lu. Evo-Harness: Context-to-harness skill compilation for self-evolving agents, 2026. URL <https://arxiv.org/abs/2608.15071>.
- [93] Guojun Zhu, Xunheng Huang, Peng Yin, Jiahui Xie, Sanguo Zhang, and Doudou Zhou. Bad genius: Counterfactual-guided harness evolution beyond task-specific shortcuts, 2026. URL <https://arxiv.org/abs/2609.18366>.

- [94] Ruiqing Yue, Yu Cui, Zhuoyu Sun, Sicheng Pan, Xianhong Xue, Tingyu Li, Ting Li, Wenzhuo Zhu, Yi Chen, Yifei Liu, Baohan Huang, Zhe Cui, Haibin Zhang, and Cong Zuo. Ecdysis: Efficient and effective training of runtime harnesses for LLM agents, 2026. URL <https://arxiv.org/abs/2609.11677>.
- [95] Geyi Yang, Zikun Qu, Xiang Li, Zhiyong Wang, Min Zhang, Shipai Zeng, and Zhongxiang Dai. GUI-HARVEST: Self-improving GUI agents through evidence-driven harness evolution, 2026. URL <https://arxiv.org/abs/2610.00948>.
- [96] Yuan Tian, Bing Hu, Hao Wang, Binghang Lu, and Fang Wu. Harness evolution hits a ceiling: When weight training should begin, 2026. URL <https://arxiv.org/abs/2610.11655>.
- [97] Xiaotian Luo, Dizhan Xue, Fengxingyu Wang, Chuanrui Hu, and Yafeng Deng. HarnessBank: Semantic gene-bank search with gated verification for agent-harness self-evolution, 2026. URL <https://arxiv.org/abs/2607.13683>.
- [98] Zhezheng Hao, Hong Wang, Jian Luo, Jianqing Zhang, Yuyan Zhou, Qiang Lin, Can Wang, Hande Dong, and Jiawei Chen. ReCreate: Reasoning and creating domain agents driven by experience. <https://aclanthology.org/2026.acl-long.1432/>, 2026.
- [99] Andrew Borthwick and Stephen Ash. RoboPhD: Self-improving text-to-SQL through autonomous agent evolution, 2026. URL <https://arxiv.org/abs/2601.01126>.
- [100] Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations*, Singapore, 2025. URL <https://openreview.net/forum?id=t9U3LW7JVX>.
- [101] Yoonho Lee, Roshen S. Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. In *Third Conference on Language Modeling (COLM 2026)*, 2026. Originally released as arXiv:2603.28052.
- [102] Peng Xia, Rujun Han, Zifeng Wang, Yanfei Chen, Yufan Zhuang, Yoonho Lee, Chengsong Huang, Han Yu, Zhongying CuiZhu, Yifei Ming, Huaxiu Yao, Burak Gokturk, Tomas Pfister, and Chen-Yu Lee. RRSI: Regularized recursive self-improvement of agent harnesses, 2026. URL <https://arxiv.org/abs/2609.24972>.
- [103] Yifan Zhang, Yutong Dai, Juntao Tan, Luyu Yang, Rishi Mullur, Thai Hoang, Zhiyuan Hu, James Zhu, Phil Mui, Silvio Savarese, Ran Xu, and Zeyuan Chen. DarwinX: Evolving agent harnesses through natural selection, 2026. URL <https://arxiv.org/abs/2608.07545>. arXiv preprint.
- [104] Wenyi Wang, Piotr Piękos, Li Nanbo, Firas Laakom, Yimeng Chen, Mateusz Ostaszewski, Mingchen Zhuge, and Jürgen Schmidhuber. Huxley-Gödel machine: Human-level coding agent development by an approximation of the optimal self-improving machine, 2025. URL <https://arxiv.org/abs/2510.21614>.
- [105] Zhaotian Weng, Antonis Antoniadis, Deepak Nathani, Zhen Zhang, Xiao Pu, and Xin Eric Wang. Group-Evolving Agents: Open-ended self-improvement via experience sharing, 2026. URL <https://arxiv.org/abs/2602.04837>.
- [106] Di Zhang. AgentDevel: Reframing self-evolving LLM agents as release engineering, 2026. URL <https://arxiv.org/abs/2601.04620>.
- [107] Anton Razzhigaev, Andrei Gritsaev, Andrei Kaznacheev, Nikita Dragunov, Roman Yampolskiy, and Andrei Kuznetsov. Ouroboros: A self-developing frontier coding agent with reviewed core evolution, 2026. arXiv:2608.08311.

- [108] Qianshu Cai, Yonggang Zhang, Xianzhang Jia, Hua Jiang Zheng, Wei Xue, Jun Song, Xinmei Tian, and Yike Guo. MOSS: Self-evolution through source-level rewriting in autonomous agent systems, 2026. URL <https://arxiv.org/abs/2605.22794>. arXiv preprint.
- [109] Jun Nie, Yonggang Zhang, Qianshu Cai, Yiu ming Cheung, Xinmei Tian, and Bo Han. EvolveNet: Collaborative harness evolution for agent self-improvement, 2026. URL <https://arxiv.org/abs/2608.04968>.
- [110] Zekai Wang, Yingqiang Ge, Zekun Wang, Hai Wang, Yuhui Xu, Joshua Frandsen, Shancong Fu, Ashia C. Wilson, and Chandan K. Reddy. VERSE: Verified self-evolving optimizer for agent harnesses, 2026. URL <https://arxiv.org/abs/2610.02616>.
- [111] Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. Gödel agent: A self-referential agent framework for recursively self-improvement. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 27890–27913, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.1354. URL <https://aclanthology.org/2025.acl-long.1354/>.
- [112] Jun Nie, Yonggang Zhang, Jun Song, Qianshu Cai, Dahai Yu, Yike Guo, Xinmei Tian, and Bo Han. TTHE: Test-time harness evolution, 2026. URL <https://arxiv.org/abs/2607.08124>.
- [113] Tianshi Xu, Huifeng Wen, and Meng Li. Adapting the interface, not the model: Runtime harness adaptation for deterministic LLM agents, 2026. URL <https://arxiv.org/abs/2605.22166>.
- [114] Erik Schluntz and Barry Zhang. Building effective AI agents. <https://www.anthropic.com/engineering/building-effective-agents>, 2024. Anthropic engineering blog, December 2024.
- [115] Yifan Shi, Wei Zhang, and Tianyi Cui. A programming paradigm for spatiotemporal composability, 2026. arXiv:2608.25512.
- [116] Tingyang Chen, Shuo Lu, Kang Zhao, Weicheng Meng, Hanlin Teng, Tianhao Li, Chao Li, Xule Liu, Jian Liang, Zhizhong Zhang, Yuan Xie, Heng Qu, Kun Shao, and Jian Luan. HarnessX: A composable, adaptive, and evolvable agent harness foundry, 2026. URL <https://arxiv.org/abs/2606.14249>. arXiv preprint.
- [117] Boxiu Li, Zimo Wen, Yijia Fan, Chuan Wen, Fan Yang, Hangxi Guo, Jiaao Wu, Jiachen Zhang, Junxiang Lei, Mukai Li, Ruize Tang, Runjing Gu, Shibo Hu, Sihan Chen, Sufeng Guo, Wanbo Zhang, Xian Zhang, Xiaoyu Chen, Xuanhe Zhou, Xuyao Huang, Yifei Gao, Yifei Shen, Yilin Chen, Yuheng Wu, Yuzhe Zhang, Zelong Zhao, and Zhijie Deng. Argus: A general-purpose agentic reasoning runtime for long-horizon tasks, 2026. URL <https://arxiv.org/abs/2608.05144>.
- [118] Wentao Zhang, Zhe Zhao, Haibin Wen, Yingcheng Wu, Cankun Guo, Ming Yin, and Bo An. Autogenesis: A self-evolving agent protocol, 2026. URL <https://arxiv.org/abs/2604.15034>.
- [119] Siwei Wu, Jincheng Ren, Yizhi Li, Haau-Sing Li, Chengran Yang, Xudong Ma, Yuxuan Zhang, Weicheng Gu, Jian Yang, Riza Batista-Navarro, Chuanyi Zhang, Xianglong Liu, Ming Zhou, Bryan Dai, and Chenghua Lin. ModularRSI: Modular and generalizable recursive harness self-improvement, 2026. URL <https://arxiv.org/abs/2609.14857>.

- [120] Yifei Shen. DSH Everything: Everything Is a Plugin and runtime self-evolution. Software repository and engineering protocols, 2026. URL <https://github.com/yshenaw/dsh-everything/tree/dade643bb09c49d290340ee01f3a00dd20bf37fb>. README, EIP self-evolution protocol, and Codex case study; accessed 14 October 2026.
- [121] Haoyang Yan, Min le Su, Hangfan Zhang, Zhanhao Li, Chen Zhang, Shao Zhang, Yang Chen, Lei Bai, and Shuyue Hu. Harness-of-Harness: Multi-day autonomous software development with continual improvement, 2026. URL <https://arxiv.org/abs/2609.01481>.
- [122] Zeyu Gan, Zixuan Gong, and Yong Liu. Harness evolution as learning: Approximation, generalization, and optimization limits of self-improving personal agents, 2026. URL <https://arxiv.org/abs/2609.36892>.
- [123] Qianshu Cai, Yonggang Zhang, Jun Nie, Maohao Ran, Huajiang Zheng, Jun Song, Xinmei Tian, Yike Guo, and Wei Xue. A theory of reliable self-evolution for agent harnesses, 2026. URL <https://arxiv.org/abs/2609.08175>.
- [124] Xi Chen, Christos Papadimitriou, and Binghui Peng. Memory bounds for continual learning. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, 2022. doi: 10.1109/FOCS54457.2022.00056.
- [125] Shai Shalev-Shwartz, Ohad Shamir, Nathan Srebro, and Karthik Sridharan. Learnability, stability and uniform convergence. *Journal of Machine Learning Research*, 11:2635–2670, 2010.
- [126] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.
- [127] Cynthia Dwork, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Roth. Generalization in adaptive data analysis and holdout reuse. *arXiv preprint arXiv:1506.02629*, 2015.
- [128] Oussama Ben Sghaier, Hao Li, Bram Adams, and Ahmed E. Hassan. Don’t blame the large language model: How agent harness evolution shapes coding agent quality, 2026. URL <https://arxiv.org/abs/2607.03691>.
- [129] Naman Vats and Oleg Golev. The scaffold effect in coding agents: Harness choice as a hidden variable in coding-agent evaluation, 2026. URL <https://arxiv.org/abs/2607.22585>.
- [130] Yike Wang, Huaisheng Zhu, Zhengyu Hu, Yige Yuan, Zhengyu Chen, Shakti Senthil, Hannaneh Hajishirzi, Yulia Tsvetkov, Noah A. Smith, Pradeep Dasigi, and Teng Xiao. Rethinking the evaluation of harness evolution for agents, 2026. URL <https://arxiv.org/abs/2607.12227>.
- [131] Varun Ursekar, Apaar Shanker, Yash Maurya, Shehab Yasser, Vijay S. Kalmath, Veronica Chatrath, and Yuan Xue. HarnessOpt-Bench: Evaluating LLMs at harness optimization, 2026. URL <https://arxiv.org/abs/2608.06301>.
- [132] Yuhao Wu, Jingyuan Zhang, Jiajun Shi, Xinping Lei, Qingshui Gu, Yuxuan Zhang, Zexuan Wang, Chen He, Chen Huang, Maojia Song, Zhiyuan Zeng, Shaowen Wang, Jinkai Liu, Yunfeng Shi, Jiaheng Liu, Shen Yan, Wenhao Huang, Ge Zhang, and Wenxuan Zhang. HarnessDev: Can LLMs create and evolve their own agent harness?, 2026. URL <https://arxiv.org/abs/2609.01437>.

- [133] Leijun Zhou, Zhihao Liu, Xiang Qu, Chenxu Liu, Yifei Liu, Yanke Yu, Jingzhe Xu, Xuejun Wu, Buyue Qian, Xi Chen, Yaowei Zheng, and Junhao Hu. GDPevo: Evaluating agent self-evolution on real business tasks, 2026. URL <https://arxiv.org/abs/2608.03764>.
- [134] Zixuan Ke, Vaidehi Patil, Haizhou Shi, Yang Li, Ye Liu, Sarath Shekkizhar, Anurag Koul, Jiayu Wang, Xuan Phi Nguyen, Semih Yavuz, Mohit Bansal, and Shafiq Joty. EvoHarnessBench: Can your agents keep pace with an evolving harness?, 2026. URL <https://arxiv.org/abs/2609.04280>.
- [135] Gavin C Cawley and Nicola LC Talbot. On over-fitting in model selection and subsequent selection bias in performance evaluation. *The Journal of Machine Learning Research*, 11: 2079–2107, 2010.
- [136] Cynthia Dwork, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Roth. The reusable holdout: Preserving validity in adaptive data analysis. *Science*, 349 (6248):636–638, 2015.
- [137] Su Wang, Pin Qian, Yifan Lin, Jingzhou Xu, Yihang Chen, Xiaochong Jiang, Lifei Liu, and Haoran Yu. Phantom Guardrails: When self-improving agent harnesses fix failures that never happened, 2026. URL <https://arxiv.org/abs/2607.13083>.
- [138] Xing Wang, Xiaoyi Zhang, and Jie Shao. Auditing harness tampering in self-improving agents, 2026. URL <https://arxiv.org/abs/2609.00069>.
- [139] Diandian Guo, Cong Cao, Fangfang Yuan, Yingqi Wang, Yueshan Wang, and Dakui Wang. Self-authored verification is unreliable in heuristic self-improving agents, 2026. URL <https://arxiv.org/abs/2607.24300>.
- [140] Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. SWE-Bench+: Enhanced coding benchmark for LLMs, 2024. URL <https://arxiv.org/abs/2410.06992>.
- [141] Michael Nguyen, Quoc Nguyen, and Paul Vuong. Recursive self-evolving agents via held-out selection, 2026. URL <https://arxiv.org/abs/2606.28374>.
- [142] Biswa Sengupta. Self-evolving agents with anytime-valid certificates, 2026. URL <https://arxiv.org/abs/2607.00871>.
- [143] 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>.
- [144] 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>.
- [145] Jiajie Wang, Yutong Zhao, Tianlin Li, Huashan Chen, Jinfu Chen, Kebin Peng, and Sen He. Agent skill evolution: How revisions affect coding agents, 2026. URL <https://arxiv.org/abs/2610.04832>.
- [146] Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. AI agents that matter. *Transactions on Machine Learning Research*, 2025.

- [147] Longju Bai, Zhemin Huang, Xingyao Wang, Jiao Sun, Rada Mihalcea, Erik Brynjolfsson, Alex Pentland, and Jiaxin Pei. How do AI agents spend your money? analyzing and predicting token consumption in agentic coding tasks, 2026. URL <https://arxiv.org/abs/2604.22750>.
- [148] Mohammad Asadolahi, Amir Amini, Samira Talebi, Amirfarhad Farhadi, and Azadeh Zamanifar. Memory reward inflation in self-improving LLM agents, 2026. URL <https://arxiv.org/abs/2608.00017>.
- [149] Xing Zhang, Guanghui Wang, Yanwei Cui, Ziyuan Li, Wei Qiu, Bing Zhu, and Peiyang He. Who grades the grader? co-evolving evaluation metrics and skills for self-improving LLM agents, 2026. URL <https://arxiv.org/abs/2607.12790>.
- [150] Xing Han Lù, Amirhossein Kazemnejad, Nicholas Meade, Arkil Patel, Dongchan Shin, Alejandra Zambrano, Karolina Stańczak, Peter Shaw, Christopher J. Pal, and Siva Reddy. AgentRewardBench: Evaluating automatic evaluations of web agent trajectories. In *Second Conference on Language Modeling (COLM 2025)*, 2025. URL <https://colm.cc/virtual/2025/poster/571>. Originally released as arXiv:2504.08942.
- [151] Jane Pan, He He, Samuel R. Bowman, and Shi Feng. Spontaneous reward hacking in iterative self-refinement, 2024. URL <https://arxiv.org/abs/2407.04549>.
- [152] Bowen Ye, Rang Li, Qibin Yang, Yuanxin Liu, Linli Yao, Hanglong Lv, Zhihui Xie, Chenxin An, Lei Li, Lingpeng Kong, Qi Liu, Zhifang Sui, and Tong Yang. Claw-eval: Towards trustworthy evaluation of autonomous agents, 2026. URL <https://arxiv.org/abs/2604.06132>. arXiv preprint.
- [153] Sayash Kapoor, Benedikt Stroebel, 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, Tengjun Jin, Yifan Mai, Yifei Zhou, Yuxuan Zhu, Rishi Bommasani, Daniel Kang, Dawn Song, Peter Henderson, Yu Su, Percy Liang, and Arvind Narayanan. Holistic agent leaderboard: The missing infrastructure for AI agent evaluation. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/a0928f924a344aaebbb7f6cd8d56e34c-Paper-Conference.pdf.
- [154] Xidong Yang, Xingyi Zhang, Wenhao Li, Wenyan Liu, Junjie Sheng, Yun Hua, Wei Yin, Tao Fang, Chuyun Shen, and Xiangfeng Wang. PATH-Bench: Path-dependent evaluation of lifelong agents, 2026. URL <https://arxiv.org/abs/2608.01149>. arXiv preprint.
- [155] Junhao Zheng, Xidi Cai, Qiuke Li, Duzhen Zhang, Zhongzhi Li, Yingying Zhang, Le Song, and Qianli Ma. LifelongAgentBench: Evaluating LLM agents as lifelong learners, 2025. URL <https://arxiv.org/abs/2505.11942>.
- [156] Tianyi Guan, Yiding Wang, Haotong Yang, Siyuan Cao, Shirui Liu, Yi Hu, Jiaqi Li, and Muhan Zhang. ContinualSkillBench: Can LLM agents truly evolve their capabilities?, 2026. URL <https://arxiv.org/abs/2608.03874>.
- [157] Shanshan Zhong, Yi Lu, Jingjie Ning, Yibing Wan, Lihan Feng, Yuyi Ao, Leonardo F. R. Ribeiro, Markus Dreyer, Sean Ammirati, and Chenyan Xiong. SkillLearnBench: Benchmarking continual learning methods for agent skill generation on real-world tasks, 2026. URL <https://arxiv.org/abs/2604.20087>.

- [158] Zexue He, Yu Wang, Churan Zhi, Yuanzhe Hu, Tzu-Ping Chen, Lang Yin, Ze Chen, Tong Arthur Wu, Siru Ouyang, Zihan Wang, Jiaxin Pei, Julian McAuley, Yejin Choi, and Alex Pentland. MemoryArena: Benchmarking agent memory in interdependent multi-session agentic tasks, 2026. URL <https://arxiv.org/abs/2602.16313>.
- [159] 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>.
- [160] Haizhong Zheng, Yizhuo Di, Ranajoy Sadhukhan, Shuwei Jin, and Beidi Chen. Serve-LearnBench: How well can agents self-improve from serving experience?, 2026. URL <https://arxiv.org/abs/2610.07792>.
- [161] Xing Zhang, Yanwei Cui, Guanghui Wang, Ziyuan Li, Wei Qiu, Bing Zhu, and Peiyang He. Ratchet: How reliable must an LLM judge be to retire a skill?, 2026. URL <https://arxiv.org/abs/2605.22148>.
- [162] Hongyi Liu, Haoyan Yang, Tao Jiang, Bo Tang, Feiyu Xiong, Yuyu Luo, and Zhiyu Li. SkillsVote: Lifecycle governance of agent skills from collection, recommendation to evolution, 2026. URL <https://arxiv.org/abs/2605.18401>.
- [163] Xing Zhang, Yanwei Cui, Guanghui Wang, Ziyuan Li, Wei Qiu, Bing Zhu, and Peiyang He. Library Drift: Diagnosing and fixing a silent failure mode in self-evolving LLM skill libraries, 2026. URL <https://arxiv.org/abs/2605.19576>.
- [164] Linfang Shang, Ming Xu, Yiding Sun, Tianle Xia, Lingxiang Hu, Lan Xu, and Ning Zheng. When self-evolution backfires: Pre-commit gating against skill contamination in LLM agents, 2026. URL <https://arxiv.org/abs/2608.05810>.
- [165] Xiaodong Wu, Yu Shi, Qi Li, Zhimin Zhao, Xiangman Li, Bram Adams, Ahmed E. Hassan, and Jianbing Ni. EVOMAL: Self-poisoning in self-evolving coding agents, 2026. URL <https://arxiv.org/abs/2608.25776>.
- [166] Ziao Zhang, Kou Shi, Shiting Huang, Avery Nie, Yu Zeng, Yiming Zhao, Zhen Fang, Qishen Su, Haibo Qiu, Wei Yang, Qingnan Ren, Shun Zou, et al. SkillFlow: Benchmarking lifelong skill discovery and evolution for autonomous agents, 2026. URL <https://arxiv.org/abs/2604.17308>.
- [167] Bowei Xia, Mengkang Hu, Shijian Wang, Jiarui Jin, Wenxiang Jiao, Yuan Lu, Kexin Li, and Ping Luo. Tool-Genesis: A task-driven tool creation benchmark for self-evolving language agent, 2026. URL <https://arxiv.org/abs/2603.05578>.
- [168] Sihang Jiang, Lipeng Ma, Zhonghua Hong, Keyi Wang, Zhiyu Lu, Tengfei Wang, Shisong Chen, Jinghao Zhang, Tianjun Pan, Weijia Li, Jiaqing Liang, and Yanghua Xiao. SEA-Eval: A benchmark for evaluating self-evolving agents beyond episodic assessment, 2026. URL <https://arxiv.org/abs/2604.08988>.
- [169] 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, 2025. URL <https://arxiv.org/abs/2512.18470>.
- [170] Parth Asawa, Christopher M. Glaze, Gabriel Orlanski, Ramya Ramakrishnan, Benji Xu, Asim Biswal, Vincent Sunn Chen, Frederic Sala, Matei Zaharia, and Joseph E. Gonzalez. Continual learning bench: Evaluating frontier AI systems in real-world stateful environments, 2026. URL <https://arxiv.org/abs/2606.05661>.

- [171] Shuhan Xue, Zixin Ding, Yichen Shen, Yinjie Wang, Zhenfei Yin, Yingcheng Wu, Yuxin Chen, Mengdi Wang, and Ling Yang. PAST-Bench: Benchmarking the foundations of recursive self-improvement in personal agents, 2026. URL <https://arxiv.org/abs/2608.04003>.
- [172] Bo Deng, Kang Zhou, Lifan Guo, Chongyang Tao, Xuanren Chen, Chenggang Xie, Renzhao Liang, Feng Chen, and Chi Zhang. FinEvo-Bench: A longitudinal benchmark for self-evolving agents in professional financial workflows, 2026. URL <https://arxiv.org/abs/2608.06144>.
- [173] Harman Singh, Anton Bakhtin, Rulin Shao, Gabriel Synnaeve, Ilia Kulikov, Rob Fergus, Sanjeev Arora, Kurt Keutzer, Jason Weston, Anuj Mahajan, and Anirudh Goyal. Agent plasticity: Measuring self-improvement through experience, 2026. URL <https://arxiv.org/abs/2610.08902>.
- [174] Xihao Piao, Zifeng Wang, and Zhen Chen. Do self-evolving skills generalize to held-out tasks?, 2026. URL <https://arxiv.org/abs/2609.39148>.
- [175] Siyuan Wang, Zhuohan Long, Zhihao Fan, Zhongyu Wei, and Xuanjing Huang. Benchmark self-evolving: A multi-agent framework for dynamic LLM evaluation, 2024. URL <https://arxiv.org/abs/2402.11443>.
- [176] Yongrui Chen, Yangyang Ma, Xiaoying Huang, Shenyu Zhang, Huajun Chen, Haofen Wang, and Guilin Qi. StressEval: Failure-driven dynamic benchmarking for knowledge-intensive reasoning in large language models, 2026. URL <https://arxiv.org/abs/2605.01939>.
- [177] Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Ben Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Sreemanti Dey, Shubh-Agrawal, Sandeep Singh Sandha, Siddhartha Naidu, Chinmay Hegde, Yann LeCun, Tom Goldstein, Willie Neiswanger, and Micah Goldblum. LiveBench: A challenging, contamination-limited LLM benchmark, 2024. URL <https://arxiv.org/abs/2406.19314>. ICLR 2025 Spotlight.
- [178] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code, 2024. URL <https://arxiv.org/abs/2403.07974>.
- [179] Zhiling Yan, Dingjie Song, Zhe Fang, Yisheng Ji, Xiang Li, Quanzheng Li, and Lichao Sun. LiveMedBench: A contamination-free medical benchmark for LLMs with automated rubric evaluation, 2026. URL <https://arxiv.org/abs/2602.10367>.
- [180] Zhiqiu Xu, Shibo Jin, Shreya Arya, and Mayur Naik. MathDuels: A self-play benchmark that grows, 2026. URL <https://arxiv.org/abs/2604.21916>. COLM 2026.
- [181] Alex Iacob, Andrej Jovanović, William F. Shen, Daniel Burkhardt, Meghdad Kurmanji, Nurbek Tastan, Lorenzo Sani, Niccolò Alberto Elia Venzani, Jiayi Nie, Ambroise Odonnat, Zeyu Cao, Bill Marino, Xinchu Qiu, Rika Antonova, and Nicholas D. Lane. The red queen Gödel machine: Co-evolving agents and their evaluators, 2026. URL <https://arxiv.org/abs/2606.26294>.
- [182] Cognition. Cascade Memories. <https://docs.devin.ai/desktop/cascade/memories>, 2026.
- [183] Cursor. Automations. <https://cursor.com/changelog/03-05-26>, 2026. Cursor Changelog, March 5, 2026.

- [184] Replit. Replit Agent Skills. <https://docs.replit.com/features/agent/skills>, 2026.
- [185] Letta. Introducing the Letta Code App. <https://www.letta.com/blog/introducing-the-letta-code-app/>, 2026.
- [186] OpenClaw. OpenClaw Skills. <https://docs.openclaw.ai/tools/skills>, 2026.
- [187] Nous Research. Hermes Agent: Skills System. <https://hermes-agent.nousresearch.com/docs/user-guide/features/skills>, 2026.
- [188] Augment Code. How We Built Memory Review. <https://www.augmentcode.com/blog/how-we-built-memory-review>, 2025.
- [189] Cognition. Devin Session Insights. <https://docs.devin.ai/product-guides/session-insights>, 2026.
- [190] Prime Intellect. Prime Agent: A Self-Improving RLM Harness. <https://github.com/PrimeIntellect-ai/prime-agent>, 2026.
- [191] Siru Ouyang, Jun Yan, Yanfei Chen, Rujun Han, Zifeng Wang, Bhavana Dalvi Mishra, Rui Meng, Chun-Liang Li, Yizhu Jiao, Kaiwen Zha, Maohao Shen, Vishy Tirumalashetty, George Lee, Jiawei Han, Tomas Pfister, and Chen-Yu Lee. SkillOS: Learning skill curation for self-evolving agents, 2026. URL <https://arxiv.org/abs/2605.06614>. arXiv preprint.
- [192] Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. Inducing programmatic skills for agentic tasks. In *Second Conference on Language Modeling (COLM 2025)*, 2025. URL <https://colm.cc/virtual/2025/poster/533>. Originally released as arXiv:2504.06821.
- [193] Doyun Kim, Chanwoo Kim, Sugyeong Eo, Yeo-Chan Yoon, and Chanjun Park. EvoSkill injection: Red-teaming autonomous skill generation and evolution in self-evolving agents, 2026. Accepted to EMNLP 2026 (per the arXiv comments field); arXiv:2608.30429.
- [194] Joar Skalse, Nikolaus Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward gaming. In *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*, volume 35, pages 9460–9471. Curran Associates, Inc., 2022. URL https://papers.nips.cc/paper_files/paper/2022/hash/3d719fee332caa23d5038b8a90e81796-Abstract-Conference.html. Originally released as arXiv:2209.13085 under the title “Defining and Characterizing Reward Hacking”.
- [195] Ömer Veysel Çağatan and Xuandong Zhao. Reward hacking in language model agents: Revisiting AI safety gridworlds, 2026. URL <https://arxiv.org/abs/2606.15385>. arXiv preprint.
- [196] Jinfeng Xu, Zheyu Chen, Ziyue Peng, Zheng Lin, Wenhao Yuan, Jian Chen, Shujie Li, and Edith Ngai. Beyond Successor Accuracy: State Retention for Recursive Self-Improvement in Recommendation. *arXiv preprint arXiv:2610.07105*, 2026. URL <https://arxiv.org/abs/2610.07105>.
- [197] Yutaro Yamada, Robert Tjarko Lange, Cong Lu, Shengran Hu, Chris Lu, Jakob Foerster, Jeff Clune, and David Ha. The AI scientist-v2: Workshop-level automated scientific discovery via agentic tree search, 2025. URL <https://arxiv.org/abs/2504.08066>.
- [198] Yanlin Fei, Nazhou Liu, Xinmiao Yu, Shaolong Chen, Lei Li, Rahul Thapa, et al. How do agents fail on AutoResearch: End-to-end diagnostic evaluation on 100 real-world frontier

- research tasks, 2026. URL <https://arxiv.org/abs/2608.14905>. AutoResearchEval; arXiv preprint.
- [199] Ao Qu, Han Zheng, Zijian Zhou, Yihao Yan, Yihong Tang, Shao Yong Ong, Fenglu Hong, Kaichen Zhou, Chonghe Jiang, Minwei Kong, Jiacheng Zhu, Xuan Jiang, Sirui Li, Cathy Wu, Bryan Kian Hsiang Low, Jinhua Zhao, and Paul Pu Liang. CORAL: Towards autonomous multi-agent evolution for open-ended discovery. In *The Third Conference on Language Modeling (COLM 2026)*, 2026. arXiv:2604.01658.
 - [200] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery, 2025. URL <https://arxiv.org/abs/2506.13131>. Google DeepMind technical report.
 - [201] PSI. The AI-native laboratory fueling the physics takeoff. <https://psi.inc/vision/>, n.d. Company vision and architecture page; accessed 22 September 2026.
 - [202] Dhruv Trehan and Paras Chopra. Why LLMs aren’t scientists yet: Lessons from four autonomous research attempts, 2026. URL <https://arxiv.org/abs/2601.03315>.
 - [203] Sébastien Bubeck, Christian Coester, Ronen Eldan, Timothy Gowers, Yin Tat Lee, Alexandru Lupsasca, Mehtaab Sawhney, Robert Scherrer, Mark Sellke, Brian K. Spears, Derya Unutmaz, Kevin Weil, Steven Yin, and Nikita Zhivotovskiy. Early science acceleration experiments with GPT-5, 2025. URL <https://arxiv.org/abs/2511.16072>.
 - [204] Hjalmar Wijk, Tao Roa Lin, Joel Becker, Sami Jawhar, Neev Parikh, Thomas Broadley, Lawrence Chan, Michael Chen, Joshua M. Clymer, Jai Dhyani, Elena Elicheva, Katharyn Garcia, Brian Goodrich, Nikola Jurkovic, Megan Kinniment, Aron Lajko, Seraphina Nix, Lucas Jun Koba Sato, William Saunders, Maksym Taran, Ben West, and Elizabeth Barnes. RE-bench: Evaluating frontier AI R&D capabilities of language model agents against human experts. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 66772–66832, 2025. URL <https://proceedings.mlr.press/v267/wijk25a.html>.
 - [205] Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 47356–47415, 2025. URL <https://proceedings.mlr.press/v267/ouyang25a.html>.
 - [206] Hyun Jung Lee, Jungtaek Kim, Jongwon Jeong, Tae-Eui Kam, Donghyun Kim, and Yong Jae Lee. EMHO: Embodied agent harness optimization via experience traces, 2026. URL <https://arxiv.org/abs/2610.08432>.
 - [207] Noematrix Team. RoboRSI: Stable, efficient, and reusable robot self-evolution in complex real-world environments. Research preview; code repository and research blog, September 2026. <https://github.com/nssmd/RoboRSI>; project page <https://lab.noematrix.ai/blog/2-roborsi-research-preview/>. Not peer-reviewed.
 - [208] Tomoyuki Kagaya, Subramanian Lakshmi, Yuxuan Lou, Thong Jing Yuan, Jayashree Karlekar, Sugiri Pranata, Natsuki Murakami, Akira Kinose, and Yang You. Memory transfer planning: LLM-driven context-aware code adaptation for robot manipulation, 2025. URL <https://arxiv.org/abs/2509.24160>.

- [209] Yuan Meng, Zhenguo Sun, Max Fest, Xiangtong Yao, Xukun Li, Zhenshan Bing, and Alois Knoll. Robotic ultra-long-horizon manipulation skills via human-guided lifelong code generation, 2025. URL <https://arxiv.org/abs/2509.18597>.
- [210] Zhengwei Xie, Zhisheng Chen, Ziyang Weng, Jinhan Li, Chenglong Li, Zikai Xiao, Jingwei Song, Jinhao Jing, Vireo Zhang, and Kun Wang. MineEvolve: Self-evolution with accumulated knowledge for long-horizon embodied Minecraft agents, 2026. URL <https://arxiv.org/abs/2603.13131>. arXiv preprint.
- [211] Konstantinos Bousmalis, Giulia Vezzani, Dushyant Rao, Coline Devin, Alex X. Lee, Maria Bauza, Todor Davchev, Yuxiang Zhou, Agrim Gupta, Akhil Raju, Antoine Laurens, Claudio Fantacci, Valentin Dalibard, Martina Zambelli, Murilo Fernandes Martins, Rugile Pevcevičiute, Michiel Blokzijl, Misha Denil, Nathan Batchelor, et al. RoboCat: A self-improving generalist agent for robotic manipulation. *Transactions on Machine Learning Research*, 2024. URL <https://arxiv.org/abs/2306.11706>. Originally released as arXiv:2306.11706.
- [212] Zeyu Shen, Haoxiang You, Yilang Liu, Zhicheng Zheng, Lihan Zha, Kashu Yamazaki, Mingtong Zhang, Suning Huang, Jiankai Sun, Qianzhong Chen, Lucy He, Kaiyuan Liu, Haoran Chang, Katerina Fragkiadaki, Dhruv Shah, Mac Schwager, Peter Henderson, Ian Abraham, and Canwen Xu. EmbodiedSWE: Coding Agents for Long Horizon Dexterous Robotics, 2026. URL <https://arxiv.org/abs/2609.27308>.
- [213] Zhexin Zhang, Shiyao Cui, Yida Lu, Jingzhuo Zhou, Junxiao Yang, Hongning Wang, and Minlie Huang. Agent-SafetyBench: Evaluating the safety of LLM agents, 2024. URL <https://arxiv.org/abs/2412.14470>.
- [214] Aarya Doshi, Yining Hong, Congying Xu, Eunsuk Kang, Alexandros Kapravelos, and Christian Kästner. Towards verifiably safe tool use for LLM agents, 2026. URL <https://arxiv.org/abs/2601.08012>. arXiv preprint.
- [215] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526, 2017.
- [216] Neel Guha, Julian Nyarko, Daniel Ho, Christopher Ré, Adam Chilton, Aditya K, Alex Chohlas-Wood, Austin Peters, Brandon Waldon, Daniel Rockmore, Diego Zambrano, et al. LegalBench: A collaboratively built benchmark for measuring legal reasoning in large language models. *Advances in Neural Information Processing Systems*, 36:44123–44279, 2023.
- [217] Tejal Patwardhan, Rachel Dias, Elizabeth Proehl, Grace Kim, Michele Wang, Olivia Watkins, Simon Fishman, Marwan Aljubeh, Phoebe Thacker, Laurance Fauconnet, Natalie Kim, Samuel Miserendino, Gildas Chabot, David Li, Patrick Chao, Michael Sharman, Alexandra Barr, Amelia Glaese, and Jerry Tworek. GDPval: Evaluating AI model performance on real-world economically valuable tasks. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/290c2430f91912204f30bbcc990fff1d-Abstract-Conference.html. Originally released as arXiv:2510.04374.
- [218] Yixing Jiang, Kameron C. Black, Gloria Geng, Danny Park, James Zou, Andrew Y. Ng, and Jonathan H. Chen. MedAgentBench: A realistic virtual EHR environment to benchmark medical LLM agents, 2025. URL <https://arxiv.org/abs/2501.14654>. arXiv preprint; associated with NEJM AI.

- [219] Samuel Schmidgall, Rojin Ziaei, Carl Harris, Eduardo Reis, Jeffrey Jopling, and Michael Moor. AgentClinic: A multimodal agent benchmark to evaluate AI in simulated clinical environments, 2024. URL <https://arxiv.org/abs/2405.07960>. arXiv preprint; later in npj Digital Medicine.
- [220] Junde Wu, Jiayuan Zhu, Minghao Hu, Fenglin Liu, and Jiazhen Pan. MedRSI: Recursive Self-Improvement for Medical Agents via Clinically Aligned Self-Evolution. *arXiv preprint arXiv:2609.24838*, 2026. URL <https://arxiv.org/abs/2609.24838>.
- [221] Shuai Shao, Qihan Ren, Dongrui Liu, Chen Qian, Boyi Wei, Dadi Guo, Jingyi Yang, Xinhao Song, Linfeng Zhang, Weinan Zhang, and Jing Shao. Your agent may misevolve: Emergent risks in self-evolving LLM agents. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/a24cd16bc361afa78e57d31d34f3d936-Abstract-Conference.html. Author order follows the ICLR proceedings record, which differs from the arXiv byline; originally released as arXiv:2509.26354.
- [222] Fei Liu and Todd Mckinnon. Beyond the hype: The efficiency-throughput gap with GitHub Copilot. *Commun. ACM*, 69(10):68–77, 2026. ISSN 0001-0782. doi: 10.1145/3797488.
- [223] Daron Acemoglu and Pascual Restrepo. The race between man and machine: Implications of technology for growth, factor shares, and employment. *American Economic Review*, 108(6):1488–1542, 2018.
- [224] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of AI on developer productivity: Evidence from GitHub Copilot. *arXiv preprint arXiv:2302.06590*, 2023.
- [225] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. Reading between the lines: Modeling user behavior and costs in AI-assisted programming. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, pages 1–16, 2024.
- [226] Nicole Forsgren, Margaret-Anne Storey, Chandra Maddila, Thomas Zimmermann, Brian Houck, and Jenna Butler. The SPACE of developer productivity: There’s more to it than you think. *Queue*, 19(1):20–48, 2021.
- [227] Judy Hanwen Shen and Alex Tamkin. How AI impacts skill formation, 2026. URL <https://arxiv.org/abs/2601.20245>.
- [228] Hamsa Bastani, Osbert Bastani, Alp Sungu, Haosen Ge, Özge Kabakcı, and Rei Mariman. Generative AI without guardrails can harm learning: Evidence from high school mathematics. *Proceedings of the National Academy of Sciences*, 122(26):e2422633122, 2025. doi: 10.1073/pnas.2422633122.
- [229] Matthias Lehmann, Philipp B. Cornelius, and Fabian J. Sting. AI meets the classroom: When do large language models harm learning?, 2024. URL <https://arxiv.org/abs/2409.09047>.
- [230] Greg Kestin, Kelly Miller, Anna Kiales, Timothy Milbourne, and Gregorio Ponti. AI tutoring outperforms in-class active learning: an RCT introducing a novel research-based design in an authentic educational setting. *Scientific Reports*, 15:17458, 2025. doi: 10.1038/s41598-025-97652-6.
- [231] Bingxi Zhao, Jiahao Zhang, Xubin Ren, Zirui Guo, Tianzhe Chu, Yi Ma, and Chao Huang. DeepTutor: Towards agentic personalized tutoring, 2026. URL <https://arxiv.org/abs/2604.26962>.

- [232] Yang Wu, Rujing Yao, Tong Zhang, Yufei Shi, Zhuoren Jiang, Zhushan Li, and Xiaozhong Liu. Teaching according to students’ aptitude: Personalized mathematics tutoring via persona-, memory-, and forgetting-aware LLMs, 2025. URL <https://arxiv.org/abs/2511.15163>.
- [233] Unggi Lee, Minchul Shin, Yeil Jeong, Sookbun Lee, Jeongsu Moon, Kyungtae Joo, Eunjoo Lee, and Hoilym Kwon. LLMs are already good tutors: Training-free prompt optimization for pedagogical math tutoring, 2026. URL <https://arxiv.org/abs/2605.27088>.
- [234] Aysa Xuemo Fan. Toward measuring AI’s effects on skill formation: The stock-formation gap, 2026. URL <https://arxiv.org/abs/2605.16283>.
- [235] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024.
- [236] Bowen Baker, Joost Huizinga, Leo Gao, Zehao Dou, Melody Y. Guan, Aleksander Madry, Wojciech Zaremba, Jakub Pachocki, and David Farhi. Monitoring reasoning models for misbehavior and the risks of promoting obfuscation, 2025. URL <https://arxiv.org/abs/2503.11926>.
- [237] Lizhi Zhang, Xin He, Dianxuan Fu, Yuyuan Feng, Jiatong Li, Qi Wang, Xin Wang, and Qinggang Zhang. SkillPoison: Progressive skill poisoning via successful experiences, 2026. URL <https://arxiv.org/abs/2610.07645>.
- [238] Saswat Das, Parvati Viswanathan, Daniel Donnelly, Chang Huang, Sahar Abdelnabi, and Ferdinando Fioretto. SEABench: Benchmarking endogenous misalignment in self-evolving agents, 2026. URL <https://arxiv.org/abs/2609.35596>.
- [239] Jon Kleinberg and Manish Raghavan. Algorithmic monoculture and social welfare. *Proceedings of the National Academy of Sciences*, 118(22):e2018340118, 2021. doi: 10.1073/pnas.2018340118.
- [240] Rishi Bommasani, Kathleen A. Creel, Ananya Kumar, Dan Jurafsky, and Percy Liang. Picking on the same person: Does algorithmic monoculture lead to outcome homogenization? In *Advances in Neural Information Processing Systems*, volume 35, 2022.
- [241] Yi Liu, Weizhe Wang, Ruitao Feng, Yao Zhang, Guangquan Xu, Gelei Deng, Yuekang Li, and Leo Zhang. Agent skills in the wild: An empirical study of security vulnerabilities at scale, 2026. URL <https://arxiv.org/abs/2601.10338>.
- [242] Yi Liu, Zhihao Chen, Yanjun Zhang, Gelei Deng, Yuekang Li, Jianting Ning, and Leo Yu Zhang. “Do not mention this to the user”: Detecting and understanding malicious agent skills in the wild. In *35th USENIX Security Symposium (USENIX Security 2026)*, 2026. arXiv:2602.06547.
- [243] Hao Song, Yiming Shen, Wenxuan Luo, Leixin Guo, Ting Chen, Jiashui Wang, Beibei Li, Xiaosong Zhang, and Jiachi Chen. Beyond the protocol: Unveiling attack vectors in the model context protocol (MCP) ecosystem, 2025. URL <https://arxiv.org/abs/2506.02040>.
- [244] Yunpeng Xiong and Ting Zhang. After the party: Growth, governance, and security scanning in the OpenClaw agent skill ecosystem. In *33rd Asia-Pacific Software Engineering Conference (APSEC 2026)*, 2026. To appear; arXiv:2609.17274.
- [245] Hechuan Guo, Yongle Hao, Yue Zhang, Minghui Xu, Peizhuo Lv, Jiezhi Chen, and Xiuzhen Cheng. A measurement study of model context protocol ecosystem, 2025. URL <https://arxiv.org/abs/2509.25292>.

- [246] Shen Dong, Shaochen Xu, Pengfei He, Yige Li, Jiliang Tang, Tianming Liu, Hui Liu, and Zhen Xiang. Memory injection attacks on LLM agents via query-only interaction, 2025. URL <https://arxiv.org/abs/2503.03704>.
- [247] Stav Cohen, Ron Bitton, and Ben Nassi. Here comes the AI worm: Unleashing zero-click worms that target GenAI-powered applications, 2024. URL <https://arxiv.org/abs/2403.02817>.
- [248] Danielle Keats Citron. Technological due process. *Wash. UL Rev.*, 85:1249, 2008.
- [249] Prannay Hebbar, Yogendra Manawat, Samuel Verboomen, Alesia Ivanova, Selvam Palani-malai, Kunal Bhatia, and Vignesh Baskaran. SIA: Self improving AI with harness & weight updates, 2026. URL <https://arxiv.org/abs/2605.27276>.
- [250] Jixuan Chen, Jiaxin Zhang, Qinyuan Ye, Yada Pruksachatkun, Haoxiang Zhang, Jingming Zhuo, Yifan Zhang, Yutong Dai, Juntao Tan, Xiangyu Peng, Silvio Savarese, Zeyuan Chen, Lianhui Qin, and Chien-Sheng Wu. CoTrace: Data recipes for training terminal agents with harness-model co-evolution, 2026. URL <https://arxiv.org/abs/2610.10426>.
- [251] Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O. Stanley. Paired open-ended trailblazer (POET): Endlessly generating increasingly complex and diverse learning environments and their solutions, 2019. URL <https://arxiv.org/abs/1901.01753>.
- [252] Chengsong Huang, Zifeng Wang, Rujun Han, Jun Yan, Yanfei Chen, Zoey CuiZhu, Ke Jiang, Peng Xia, Han Yu, Yufan Zhuang, Yifei Ming, Jiaqi Pan, Bhavana Dalvi Mishra, Jiaxin Huang, Burak Gokturk, Tomas Pfister, and Chen-Yu Lee. EnvHarness: Awakening static worlds for agent learning, 2026. URL <https://arxiv.org/abs/2608.19880>.
- [253] Yixuan Li, Yiyun Zhou, Yao Long Teng, Fuchao Yang, Yanchen Deng, Zhiyi Lyu, Xuyu Dong, Feng Chen, and Bo An. Finding the right fit: Model-harness interactions across agent tasks, 2026. URL <https://arxiv.org/abs/2610.00917>.
- [254] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7 (PLDI):1946–1969, 2023. doi: 10.1145/3591300.
- [255] Zhening Li, Joshua Liu, Mateja Vukelic, Nicole Shen, Supriya Lall, Amitayush Thakur, Alex Zhang, Omar Khattab, Jonathan Light, and Armando Solar-Lezama. Harness as a Language: A Minimalist Agent Framework With Maximal Expressivity. *arXiv preprint arXiv:2609.26891*, 2026. URL <https://arxiv.org/abs/2609.26891>.
- [256] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, pages 625–635, 2021. doi: 10.1007/978-3-030-79876-5_37.
- [257] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J. Prenger, and Animashree Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Advances in Neural Information Processing Systems*, volume 36, pages 21573–21612, 2023. doi: 10.52202/075280-0944. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/4441469427094f8873d0fecb0c4e1cee-Abstract-Datasets_and_Benchmarks.html.
- [258] Haiming Wang, Huajian Xin, Chuanyang Zheng, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, Jian Yin, Zhenguo Li, and Xiaodan Liang. LEGO-prover: Neural theorem proving with growing libraries. In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=3f5PALef5B>.

- [259] Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/444a3737adaee10d86ad2ef5f74468e6-Paper-Conference.pdf.
- [260] Avrim Blum and Moritz Hardt. The ladder: A reliable leaderboard for machine learning competitions. In *International Conference on Machine Learning*, volume 37, pages 1006–1014. PMLR, 2015.
- [261] Tianhao Wu, Weizhe Yuan, Olga Golovneva, Jing Xu, Yuandong Tian, Jiantao Jiao, Jason Weston, and Sainbayar Sukhbaatar. Meta-rewarding language models: Self-improving alignment with LLM-as-a-Meta-Judge, 2024. URL <https://arxiv.org/abs/2407.19594>.
- [262] Rui Meng, Bhavana Dalvi Mishra, Jiefeng Chen, Chun-Liang Li, Palash Goyal, Mihir Parmar, Yiwen Song, Yale Song, Rajarishi Sinha, Parthasarathy Ranganathan, Burak Gokturk, Jinsung Yoon, and Tomas Pfister. ScientistOne: Towards human-level autonomous research via chain-of-evidence, 2026. URL <https://arxiv.org/abs/2605.26340>. arXiv preprint.