miniFlux: Exploring Agent-Directed Harness Evolution for Online Adaptation

Anonymous authors
Paper under double-blind review

Abstract

Agent harnesses have become a crucial factor in agent performance, shaping how models process context, take actions, and receive feedback. However, the support an agent needs can vary across tasks and evolve during execution. Existing approaches to automated harness evolution typically operate offline, relying on fixed task executions that may not capture diverse, emerging needs during deployment. We introduce miniFlux, an experimental framework for agent-directed online harness evolution within each task execution. Starting from a minimal harness (mini), the same model alternates between a main executor that advances the task and an evolver that adapts the active harness based on recent execution evidence. Triggered by periodic checks or executor requests, the evolver uses a bounded tool-use loop to revise runtime prompt guidance, tools, middleware, and context management. Component updates that pass interface and loading checks take effect through hot reload, preserving the progress without resetting the task. Evaluations on ARC-AGI-3, Tetris, and GDPval tasks show that miniFlux achieves higher mean task scores and improved action-level efficiency than the fixed mini baseline across evaluated models, and outperforms Codex in some evaluated settings. Cost analysis shows that online evolution adds relatively small overhead, representing less than 30% of the main execution cost in most evaluated settings. Evolution analyses illustrate that agents create and adapt task-specific components, such as analysis tools and validation middleware, to support information processing and task delivery. The ablation of evolution triggers shows that combining scheduled and agent-requested updates achieves higher mean performance than either mode alone. These findings suggest that adapting the harness online to emerging needs is a promising direction for improving agent performance.

1 Introduction

Agent harnesses transform large language models (LLMs) from single-turn question-answering bots into interactive agents capable of acting in environments and solving real-world problems. Modern harnesses provide components such as prompts, tools, context management, and middleware that hooks around model and tool calls. Recent work shows that harness design can substantially affect model performance (Wu et al., 2026; Merrill et al., 2026; Yao et al., 2026), making it essentially a crucial part of task solving, rather than a neutral wrapper around the model.

Improving agent performance on a harness typically involves *harness engineering*: human engineers conduct controlled experiments and refine harness components based on the empirical observations (Rajasekaran, 2026; Trivedy, 2026). However, manual harness engineering is labor-intensive and difficult to scale across diverse tasks and rapidly evolving base models. Recent studies seek to automate this process by using LLM agents to inspect and revise harnesses (Lee et al., 2026; Lin et al., 2026; Zhang et al., 2026a). Although these approaches can improve agent performance on specific tasks, repeated execution and analysis can be impractical when tasks are expensive to rerun or must be completed in a single attempt. Moreover, a harness optimized offline on a development set may not generalize well to heterogeneous, evolving deployment environments. Our intuition is that the support an agent needs from its harness can vary across tasks and even across different stages of the same task. For example, an unfamiliar game may initially require exploration and later benefit from a tool for repeated actions. A document task may require output validation checks only after

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

the deliverable requirements and target files have been clarified. Such needs often become apparent only through interaction with the environment, motivating dynamic harness updates.

In this work, we introduce miniFlux, an experimental framework for *online harness evolution* that interleaves automated harness engineering with task execution to address task-specific, execution-dependent needs within a single task attempt. Built on mini-swe-agent (mini) (Yang et al., 2024), miniFlux starts with a minimal entry point and continuously evolves itself. During execution, the same frozen model alternates between two roles: a main executor that works on the user's task and a harness evolver that reviews recent evidence and uses harness-management tools to inspect and update harness components. Without final outcomes, the evolver uses the recent execution trajectory, records of previous updates, and optional executor requests to identify emerging needs and address runtime issues and limitations. Proposed changes that pass continuous integration (CI) checks take effect in the ongoing task through hot reloading. Harness updates remain local to each task attempt and do not persist across attempts, so each new attempt starts from the same clean harness. This design allows the agent to adapt its harness flexibly without repeatedly running the same tasks to select among candidate harnesses or requiring human engineers to design and test each update.

We evaluate miniFlux primarily against the fixed mini harness from which it starts. Across experiments with DeepSeek-V4-Flash and GLM-5.2, miniFlux improves overall mean performance across diverse scenarios, including the gaming environments ARC-AGI-3 and Tetris and the professional-work benchmark GDPval. Improvements in the game action and agent step metrics further indicate greater action-level efficiency. Comparisons with Codex provide additional context: miniFlux approaches or exceeds its performance under several evaluated conditions, while using a substantially simpler harness design and less token cost. To understand these gains, we analyze the evolution process and find that agents proactively create and install task-dependent components, such as analyzer tools and output validation middleware, that help them process task information and complete tasks successfully. Cost analysis shows that evolution incurs less than 30% of main execution in five of six conditions. Evolution trigger-mode analysis indicates that combining scheduled and agent-requested updates better supports stable evolution and timely repair.

Our contributions are threefold: (i) We introduce miniFlux, a framework for online harness evolution in which the same base model alternates between task execution and harness adaptation within each task attempt. (ii) We empirically validate miniFlux on ARC-AGI-3, Tetris, and GDPval, demonstrating higher mean task performance and greater execution efficiency than the static harness baseline, with modest evolution overhead. (iii) We analyze the task-dependent components agents create and adapt, and show that combining fixed-interval and executor-requested triggers achieves higher mean performance than either mechanism alone.

2 Related Work

Offline Harness Evolution. Harness evolution traditionally relies on manual engineering: human developers inspect model behavior and execution failures, revise prompts, tools, and control flow to address observed patterns, and compare candidate designs through controlled experiments (Rajasekaran, 2026; Trivedy, 2026). Recent work automates this development loop by assigning diagnosis and revision jobs to LLM agents. Meta-Harness searches harness code using prior candidates, scores, and execution traces (Lee et al., 2026), and Agentic Harness Engineering (Lin et al., 2026) makes editable components and trajectory evidence explicit and checks predicted effects against subsequent task outcomes. HarnessX (Chen et al., 2026) supports such evolution through composable runtime primitives and a trace-driven evolution engine. Self-Harness (Zhang et al., 2026a) mines model-specific weaknesses and subjects proposed changes to regression tests. HarnessEvolve (Jiang et al., 2026) diagnoses failures against reference trajectories generated with ground-truth answers, then applies quality and performance gates. Together, these studies develop mechanisms for revising harnesses based on previous task executions. However, repeated executions can be costly, and harnesses optimized offline may not address the diverse needs that emerge at test time.

Online Harness Evolution. Recent work further brings harness evolution into the deployment stage. TTHE (Nie et al., 2026) evolves candidate harnesses on unlabeled test batches and selects updates using execution-derived proxy signals. More closely related to our setting, Continual Harness (Karten et al., 2026) refines prompts, sub-agents, skills, and memory within a single persistent

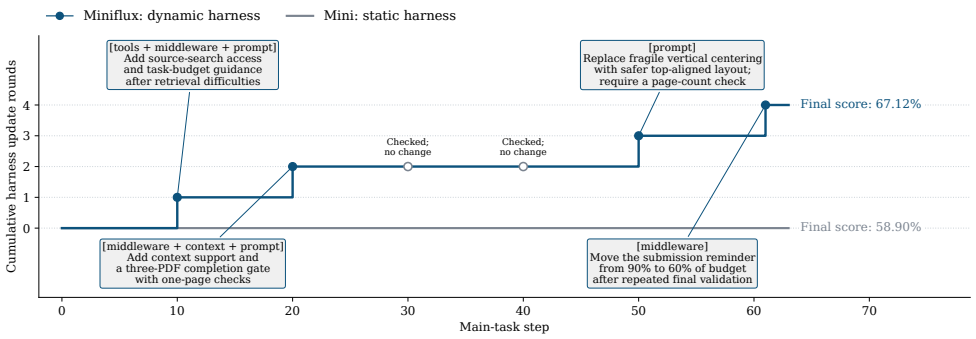


Figure 1: **MiniFlux online harness evolution on a GDPVal task with GLM-5.2**. At step 10, miniFlux installs a web tool to recover failed retrieval and configures budget middleware for the 100-step limit. At step 20, it enables a context compressor and customizes output validation middleware to check three PDFs and submission.txt. Checks at steps 30 and 40 make no changes. At step 50, a prompt update recommends removing extra vertical spacing before the checklist to resolve page overflow; all three PDFs pass single-page validation at step 52. At step 61, it advances the submission reminder from 90% to 60% of the step budget, prompting the agent to submit. Final scores are 67.12% for the selected miniFlux run versus 58.90% for the static mini runs.

run. Its harness is developed and evaluated for Pokémon gameplay, with refinement organized into four prescribed component-specific passes over trajectory evidence. We share its motivation for online adaptation without task resets, while targeting a general-purpose agent harness and a more flexible evolver design. Within a bounded tool-use loop, the evolver chooses which harness components to inspect and update, using feedback from each operation to guide subsequent decisions. It can make multiple updates or finish without changes when no intervention is warranted. Thus, harness refinement proceeds through an adaptive sequence of inspection and editing decisions rather than a fixed LLM call for each component update.

Self-evolving Agents. Harness evolution can be regarded as a subcategory in the broader field of self-improving agents, whose approaches can be distinguished by the evolution targets. Memory-based adaptation includes retaining verbal feedback across trials (Shinn et al., 2023) and dynamically organizing memories (Xu et al., 2025). Evolution can also involve discovering and refining reusable skills (Zheng et al., 2025), constructing and revising task procedures (Lu et al., 2026), updating model weights through test-time training (Yuksekgonul et al., 2026), and modifying agent architectures through code revisions (Wang et al., 2026; Zhang et al., 2026b). These categories overlap: memory and skills can also be harness components. This broader line of research also connects to the emerging field of recursive self-improvement (RSI), where successive revisions shape later improvement cycles. Dream-RSI (Zheng et al., 2026), for example, iteratively refines exploration policies using replay of recorded discovery histories.

3 METHOD

We first introduce the key design elements of miniFlux: four evolvable component classes, pluggable components, a hot-reload kernel, and a role-switching mechanism (§ 3.1). We then describe how the main agent executes tasks and requests harness updates (§ 3.2), followed by the execution evidence, harness update tools, and CI checks that support the evolver's adaptation process (§ 3.3).

3.1 MINIFLUX: A MODULAR, EVOLVABLE HARNESS

An agent harness H provides a scaffolding layer between a base model M and the task environment. Prior work explores different harness components as targets for evolution (Lin et al., 2026; Karten et al., 2026; Zhang et al., 2026a). Inspired by Trivedy (2026), miniFlux builds on the minimal interface of mini-swe-agent and exposes four evolvable classes: (1) **Runtime prompt** provides a concise hand-off from the evolver, informing the main agent of harness updates and guiding subsequent actions through an appended runtime message while leaving the system prompt unchanged. (2) **Tools**

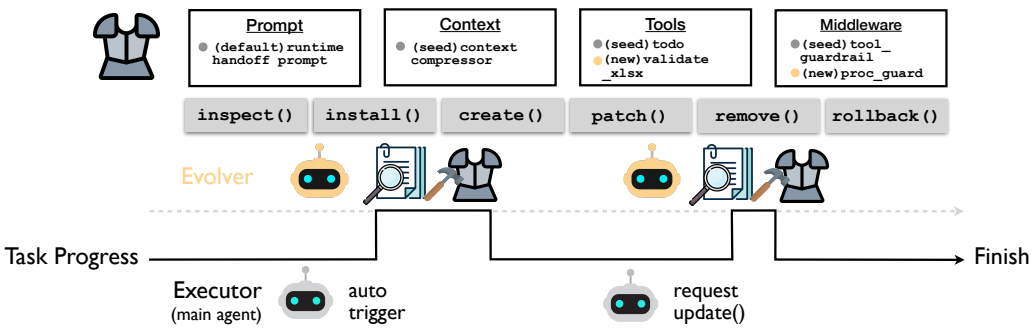


Figure 2: **Overview of miniFlux workflow**. Task execution alternates with bounded evolution cycles, triggered automatically or requested by the executor. The evolver inspects recent execution and updates runtime guidance, context compression, tools, or middleware. Execution then continues with the updated harness. Component names illustrate available or newly authored support; they do not imply that all components are enabled initially.

encapsulate operations that the main agent can invoke. (3) **Middleware** introduces policies, checks, or feedback around model queries and tool executions. (4) **Context management** controls the message history presented to the model and compacts accumulated context when a specified threshold is reached. Together, these four classes enable the agent to adapt its execution strategy, available actions, execution policies, and working context.

The pluggable component design and hot-reload kernel allow individual harness components to evolve and function without restarting the core agent loop. Tools, middleware, and context management are implemented as standalone source components that can be dynamically loaded, modified, or removed. The initial seed repository provides generic components for file operations, web search, budget feedback, and context compression under each class, yet they are not installed by default (Appendix D). The evolver can selectively install and adapt these components or create new ones as task needs emerge. Harness update requests and runtime prompt guidance connect the two roles: the main agent can explain its needs when requesting an update, and the evolver can provide an evolution digest and guidance when handing execution back. The miniFlux kernel invokes active components at the appropriate execution hooks and loads accepted changes for subsequent steps, preserving task progress while making revised parts effective.

Role-switching alternates the same underlying model M between task execution and harness evolution, with separate system prompts, tool access, and conversations for each role. The main agent advances the task using the active harness. At a scheduled or agent-requested trigger, task execution pauses while the evolver reviews the current execution context and uses harness-management tools to adapt the harness within a bounded budget. The main agent then resumes with any accepted changes and an accompanying runtime message; if no changes are made, it continues with the existing harness. This separation keeps each role focused on its objective while preserving the task state across evolution cycles. Figure 2 illustrates this alternation.

3.2 Main Agent: Task Execution

The main agent starts from the mini harness with only a `bash()` tool and directly works in the task environment to carry out the task specified by the user. When agent-requested evolution mode is enabled, it also receives a `harness_update()` tool, which requests an evolution cycle after the current step and accepts an optional reason. The request identifies a need for support; the evolver decides how to intervene. After an update, the main agent receives a runtime message containing a compact digest of changed targets and the evolver's hand-off, then continues from its current task state with the updated harness.

3.3 Evolver Agent: Evidence-Guided Adaptation

The evolver operates in a separate conversation with a dedicated system prompt and harness-management tools. Its task prompt combines the original user prompt, recent trajectory window,

Algorithm 1 miniFlux online harness evolution

Require: model M, task prompt P_{main}, environment E, base harness H_0, update interval K, main-agent budget N, evolver harness H_e, evolver round budget B

$H \leftarrow H_0$, $\mathcal{T} \leftarrow [P_{\text{main}}]$, $\mathcal{U} \leftarrow []$, $t \leftarrow 0$, `next_evo` $\leftarrow K$
while $t < N$ **do** ▷ main agent loop
  $\tau \leftarrow \textsc{MainAgentStep}(M, H, E, \mathcal{T})$
  $\textsc{Append}(\mathcal{T}, \tau)$; $t \leftarrow t + 1$
  if $\textsc{Terminated}(\tau)$ **then break**
  if update $\in \textsc{Actions}(\tau)$ **or** $t \geq$ `next_evo` **then**
    $P_{\text{evo}} \leftarrow \textsc{Template}(P_{\text{main}}, \mathcal{T}_{[-K:]}, \mathcal{U}, \text{update}, B)$; $\mathcal{T}_{\text{evo}} \leftarrow [P_{\text{evo}}]$
    for $b = 1, \ldots, B$ **do** ▷ evolver agent loop
      $(\tau_{\text{evo}}, H) \leftarrow \textsc{EvolverStep}(M, H_e, H, \mathcal{T}_{\text{evo}})$ ▷ harness updates
      $\textsc{Append}(\mathcal{T}_{\text{evo}}, \tau_{\text{evo}})$
      if `finish_evolution` $\in \textsc{Actions}(\tau_{\text{evo}})$ **then break**
    end for
    $\textsc{Append}(\mathcal{U}, \textsc{Digest}(\mathcal{T}_{\text{evo}}))$; $\textsc{Append}(\mathcal{T}, \textsc{Handoff}(\mathcal{T}_{\text{evo}}))$; `next_evo` $\leftarrow t + K$
  end if
end while

compact records of previous updates, the current trigger step and evolution budget, and optionally the main agent's stated update reasons. The prompt templates are provided in Appendix F. Without access to the final task outcome, the evolver uses execution feedback to identify needs and revise the harness, and can also finish without changes when the evidence does not justify an intervention.

Table 1 summarizes the harness update interface available to the evolver. Within a fixed budget of model calls, the evolver can inspect the harness, install or adapt seed components (Appendix D), or create new components from scaffolds. The framework provides immediate feedback on the outcome of each harness update operation, guiding subsequent edits and retries within the same evolution cycle. Source edits apply to task-local copies of components, and accepted changes take effect when the main agent resumes execution. The runtime prompt is maintained separately in memory and delivered to the main agent.

Table 1: Harness update tools available to the evolver.

Tool name	Brief description
`harness_inspect`	Inspect available components, source, and manifests.
`harness_create`	Scaffold a task-local component.
`harness_install`	Validate and load a component.
`harness_patch`	Edit component source.
`harness_remove`	Unload a component.
`harness_rollback`	Restore a component state.
`finish_evolution`	End the current evolution cycle.

When installing a component or accepting an edit to an active component, the framework runs fixed continuous integration (CI) checks for manifest validity, component identity, successful source import, and interface compatibility. For example, tool components must expose exactly one tool with a valid schema and no name collision; middleware and context components must implement their declared hooks. A failed operation restores its prior state and returns error feedback, while accepted changes are recorded for hot reload and explicit rollback. Acceptance is per operation: a later failure or an exhausted evolution budget does not discard earlier accepted updates. CI establishes that a component satisfies the supported interface, but does not guarantee functional correctness. The practical effects for the evolved components become observable through subsequent main-agent execution and can incentivize later evolution cycles. Algorithm 1 summarizes the complete workflow with two nested loops and Figure 1 illustrates this process with a concrete example.

4 Experiments

4.1 Experimental Setup

Benchmarks. The evaluation covers three benchmarks, including two gaming environments and one professional task setting, selected for their challenging and diverse nature. *ARC-AGI-3* (ARC Prize Foundation, 2026) comprises 25 interactive games with 183 levels, where agents must infer

Table 2: Model performance with different harnesses across three benchmarks. Entries with ± report means and standard deviations over 3 runs. "Act" counts game actions in ARC-AGI-3 and Tetris; "Step" counts agent turns in GDPval. Best values within each model are bold.

		ARC-AGI-3 (text)			Tetris (text)		GDPval-Rubric	
Model	**Harness**	**RHAE (↑)**	**Levels (↑)**	**Act/Level (↓)**	**Score (↑)**	**Act/Score (↓)**	**Score (↑)**	**Step/task (↓)**
DeepSeek-V4-Flash	Codex	1.79 ± 0.80	17.67 ± 4.73	449.64	**35.78** ± 3.40	**0.72**	**79.66** ± 1.23	35.79
	mini	2.99 ± 0.34	20.33 ± 3.21	338.48	18.44 ± 2.50	1.02	76.11 ± 0.64	45.08
	miniflux	**3.32** ± 1.72	**24.33** ± 7.37	**225.70**	27.00 ± 8.08	0.89	78.97 ± 0.42	**34.61**
	Δ	+0.34	+4.00	-112.78	+8.56	-0.13	+2.86	-10.46
GLM-5.2	Codex	**2.30** ± 1.19	14.67 ± 1.15	**415.68**	**122.56** ± 46.83	**0.39**	78.20 ± 4.24	34.27
	mini	1.18 ± 1.48	8.33 ± 5.86	670.88	82.22 ± 26.20	0.46	81.79 ± 0.76	34.99
	miniflux	2.17 ± 0.63	**15.33** ± 5.86	431.24	104.78 ± 20.71	0.40	**82.86** ± 1.64	**31.63**
	Δ	+0.99	+7.00	-239.64	+22.56	-0.06	+1.06	-3.36

hidden rules through interaction and complete as many levels as possible within a limited action budget. *Tetris* (Hu et al., 2026) requires agents to position and rotate falling pieces to clear complete rows, accumulate score, and prevent the stack from reaching the top of the board. We follow the official evaluation protocol using seeds 42–44. For *GDPval* (Patwardhan et al., 2026), agents receive professional task instructions and reference files and produce deliverables such as Word documents, and PDFs. We retain 44 tasks from the 50-task subset curated by OpenSpace Team@HKUDS (2026), excluding six tasks that require multimodal processing or contain broken reference files.

Models and harnesses. We evaluate DeepSeek-V4-Flash-0731 (Xu et al., 2026) and GLM-5.2-FP8 (Zeng et al., 2026) (served locally with a 400K context window), with their default thinking effort and temperatures. Mini-swe-agent (mini) serves as the static baseline, and miniFlux starts from its minimal interface (Section 3). Codex provides a more extensively engineered harness reference, with a detailed system prompt and advanced execution tools. Within each model condition, miniFlux uses the same underlying model for task execution and harness evolution. Our default configuration combines automatic checks every 20 main-agent steps with executor-requested evolution, allowing up to 20 evolver model calls per evolution cycle. For GDPval, we shorten the automatic check interval to 10 steps to account for the shorter average task length.

Metrics. For ARC-AGI-3, we report the benchmark's official RHAE score, completed levels, and game actions per completed level, with the last metric computed as total game actions divided by total completed levels. For Tetris, we report native game score and actions per score, computed as total game actions divided by total score. In both game environments, game actions are exposed through CLI commands to support evaluation across harnesses. We count game actions to measure direct interactions with the game, while agents may take additional reasoning or analysis steps between game actions. For GDPval, we report mean rubric score as a percentage of maximum achievable rubric score across all task attempts and main-agent steps per task to measure the execution efficiency. To reduce variance, all results are averaged over 3 independent runs. Additional metric scoring details are provided in Appendix B.

4.2 Main Results

Online evolution improves on the static mini baseline. As shown in Table 2, miniFlux achieves higher mean task scores than mini in all six model–benchmark conditions. On ARC-AGI-3, RHAE improves by 0.34 for DeepSeek and 0.99 for GLM, alongside 4.00 and 7.00 additional completed levels, respectively. On Tetris, mean scores increase by 8.56 for DeepSeek and 22.56 for GLM. On GDPval, rubric scores improve by 2.86 and 1.06 percentage points, respectively. These consistent gains suggest that agent-directed online evolution can adapt the harness to better support final task outcomes across diverse evaluation settings.

The performance gains are accompanied by improvements in action efficiency. Actions per completed level on ARC-AGI-3 decrease by 33.3% for DeepSeek and 35.7% for GLM, while Tetris actions per score decrease by 12.7% and 13.0%, respectively. On GDPval, miniFlux uses 10.46 fewer main-agent steps per task for DeepSeek and 3.36 fewer for GLM. These results suggest that

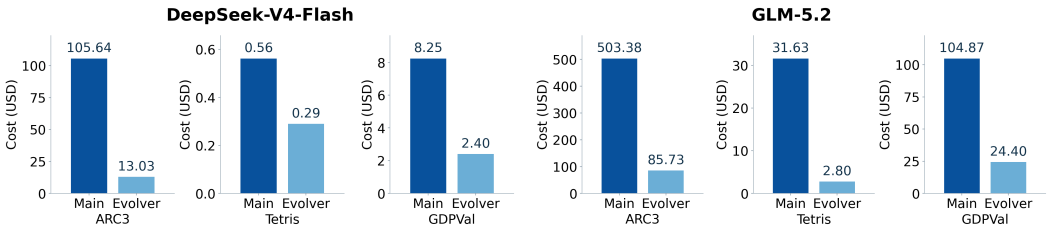


Figure 3: Comparison of main-agent and evolver costs for miniFlux, summed over three repeat runs.

online harness adaptation helps agents act more effectively, reducing redundant interactions that may arise from insufficiently tailored harness support.

miniFlux leads Codex in certain settings. Despite starting from a minimal interface, miniFlux outperforms the more extensively engineered Codex harness in several settings. To contextualize their implementation sizes, the core agent harness contains approximately 100 lines of code (LOC) for mini, 1K LOC for the evolved miniFlux harness with DeepSeek on ARC-AGI-3, and 80K LOC for Codex. With DeepSeek on ARC-AGI-3, miniFlux achieves higher RHAE and completes more levels than Codex. With GLM on GDPval, miniFlux achieves a higher rubric score while requiring fewer agent steps per task. Codex nevertheless retains higher scores and lower actions per score on Tetris with both models. These suggest that online adaptation can yield competitive performance from a substantially smaller harness, although the advantage depends on the model and task.

4.3 Evolution Cost

We examine the cost of online evolution from two perspectives. First, we compare evolver costs with main-agent costs across all three benchmarks to quantify the additional overhead. We then compare different agent harnesses to assess the token use of the evolved harness for main task execution.

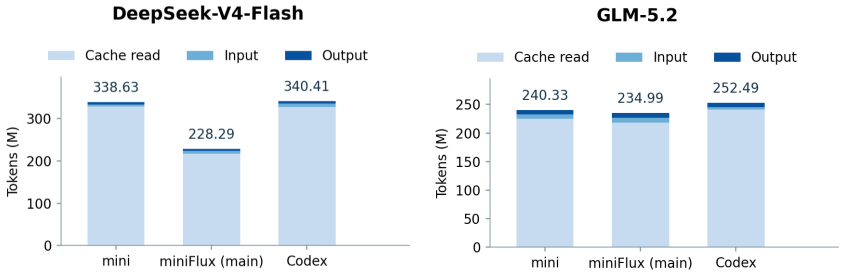


Figure 4: Comparison of main-agent token use on GDPval summed over three repeat runs.

Online evolution adds a relatively small cost. Figure 3 compares the recorded dollar costs of the main agent and evolver within miniFlux. We measure evolution overhead as evolver cost divided by main-agent cost. Using the values shown in the figure, this overhead is approximately 12.3%, 51.8%, and 29.1% for DeepSeek on ARC-AGI-3, Tetris, and GDPval, respectively. For GLM, the corresponding values are 17.0%, 8.9%, and 23.3%. Evolution therefore costs less than 30% of main execution in five of the six conditions. DeepSeek on Tetris has a higher ratio because it does not progress far into the game, keeping main execution short and its cost low. In all six conditions, the average evolution cost per run is far below the average cost of one main-agent run.

Evolved harnesses can reduce execution token use. Beyond its added cost, evolution can reduce the tokens needed for main-agent execution, although these savings vary across conditions. Figure 4 focuses on GDPval because it contains fixed tasks. In contrast, ARC-AGI-3 and Tetris are progressive games, where advancing further can consume more tokens (Appendix C).

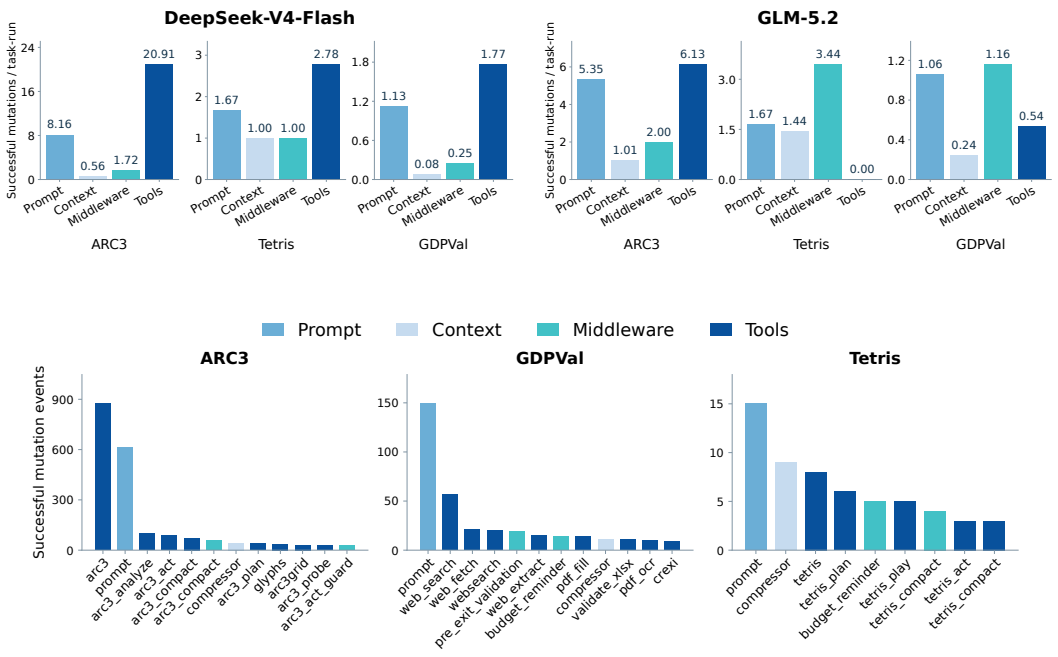


Figure 5: Harness evolution patterns across models and benchmarks on three repeat runs. Top: mean successful mutations per task-run, grouped by component class. Bottom: the most frequently evolved individual components for DeepSeek-V4-Flash, where each panel shows the top 12 components, or all available components if fewer than 12.

On GDPval, miniFlux reduces total main agent tokens from 338.63M to 228.29M for DeepSeek, a 32.6% reduction relative to mini, and from 240.33M to 234.99M for GLM, a 2.2% reduction. Both totals are also below Codex, which uses 340.41M and 252.49M tokens, respectively. Together with the higher scores than mini in Table 2, these results show that the evolved harness can improve task outcomes while using fewer execution tokens. Nevertheless, the initial harness and the nature of the task may still affect how much room evolution has to improve execution cost.

5 Analysis

We further examine what the miniFlux actually evolves and which mechanisms contribute to its effectiveness. In § 5.1, we analyze evolution at progressively finer levels: component classes, individual components, and key changes. In § 5.2, we examine how triggering modes affect evolution frequency and final performance, finding that the mixed triggering mode performs the best.

5.1 Evolution Analysis

Evolution paths vary across models and tasks, with prompts and tools frequently updated. Figure 5 (top) summarizes successful mutations per task-run, averaged over three repeats. On ARC-AGI-3, tools receive the most updates for both DeepSeek-V4-Flash and GLM-5.2, followed by prompts. DeepSeek also favors tools on Tetris and GDPval-Rubric, whereas GLM updates middleware most frequently on these benchmarks. The component-level counts in Figure 5 (bottom) further reveal task-specific concentrations. For DeepSeek, tools.arc3 dominates ARC-AGI-3 with 873 mutations. On GDPval-Rubric, prompt updates lead with 149 mutations, followed by tools.web_search with 57; on Tetris, prompts and context.compressor receive most mutations. These patterns suggest that harness evolution responds to task-specific needs: the components selected for adaptation vary across models and tasks, while prompts and tools remain common choices.

Effective parsing, analysis and validation components emerge for task-specific uses. To identify evolved components that may support more effective task completion, we select, for each com-

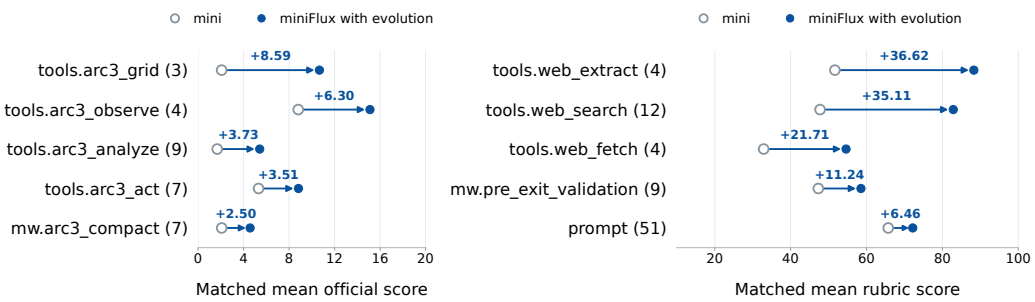


Figure 6: Component-wise task-matched scores for DeepSeek-V4-Flash on ARC-AGI-3 (left) and GDPval-Rubric (right). Parentheses count miniFlux runs containing that evolution. Each panel shows top 5 components with the largest positive score associations. "mw" denotes middleware.

ponent, all miniFlux runs in which it emerges and compare their mean score with that of baseline mini runs on the same tasks. Figure 6 highlights representative components associated with the largest improved performance. On ARC-AGI-3, benchmark-specific parsing and analysis tools, including arc3_grid and arc3_analyze, are associated with notable gains. These tools provide specialized utilities for interpreting environment observations; we present a concrete implementation of arc3_analyze in Appendix E. On GDPval, web tools and validation middleware are associated with large gains. Further inspection of the execution traces suggests that web tools streamline retrieval and help agents avoid stalling on web operations, effectively preventing from budget exhaustion. Validation middleware prompts agents to recheck their outputs before submission, helping catch errors and omissions. Together, these examples suggest that miniFlux can create new components and adapt seed components to support task-specific information processing and validation.

5.2 Triggering Mode Analysis

To enter the evolver role and update the harness, miniFlux combines automatic checks at fixed intervals with update requests from the main agent. We ablate this design in Table 3, which shows that mixed triggering achieves the highest mean score with the highest evolution frequency.

Table 3: Harness update trigger-mode ablation on ARC-AGI-3 with GLM-5.2.

Mode	Evolution Frequency	Final Score (with SD)
Mixed mode (default)	**5.59**	**2.17 ± 0.63**
Auto-update only	4.85	1.14 ± 0.15
Self-request only	0.11	0.84 ± 0.57

Relying solely on the main agent yields only eight update requests across 75 task-runs (0.11 cycles per task) and the lowest mean score. This low frequency suggests that the agent rarely initiates proactive harness improvements while solving tasks. Periodic checks provide regular opportunities for adaptation, but still underperform the mixed mode. Our further analysis reveals that the two mechanisms can complement each other: self-requests allow the main agent to report concrete limitations after getting stuck, and to request repairs to the newly evolved components (Appendix G). Such requests enable timely intervention without waiting for the next scheduled cycle.

6 Conclusion

In this work, we introduced miniFlux, an automated framework that uses the same model to execute tasks and adapt its harness based on ongoing execution feedback within a single attempt, without access to final outcomes or a fixed development set. Across DeepSeek-V4-Flash and GLM-5.2, miniFlux improves mean task performance and action-level efficiency over the fixed mini baseline with modest evolution overhead. Analyses reveal that effective task-dependent component adaptations emerge during execution, while periodic checks and executor-requested updates complement each other. These findings suggest that online harness evolution can improve task performance by adapting the agent's support mechanisms to needs that emerge during execution.

AI USE STATEMENT

In this work, we used generative AI tools to design or provide feedback on research methodology and experiments, implement methods, assist with translation, support qualitative and thematic data analysis, and interpret results. We did not use generative AI tools to develop theoretical models or conceptual frameworks, propose or refine hypotheses, or clean and reformat datasets. Generating synthetic datasets, formulating mathematical claims, providing critical ingredients for proving mathematical claims, and assisting in writing proofs are not applicable to this work. Additionally, we used generative AI tools to create or modify scientific figures and images, create or edit software code, draft parts of the paper, summarize or analyze existing literature, brainstorm ideas, source and search for information, identify relevant literature, and propose a title or keywords for the paper.

We have reviewed all AI-assisted work, including checking the generated code and verifying the results and their interpretations for correctness. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

To facilitate reproducibility, we explain our method in the Section 3, describing the underlying algorithm of our approach in Algorithm 1. We also provide the experimental configurations in Appendix B, including the agent harness versions, benchmark settings, and the judge model used in our experiments. The complete source code, including the evaluation code used in our experiments, is attached in the supplementary materials to support independent implementation checks and reproduction of the reported results.

REFERENCES

ARC Prize Foundation. Arc-agi-3: A new challenge for frontier agentic intelligence. *arXiv preprint arXiv:2603.24621*, 2026.

Tingyang Chen, Shuo Lu, Kang Zhao, Weicheng Meng, Hanlin Teng, Tianhao Li, Chao Li, Xule Liu, Jian Liang, Zhizhong Zhang, et al. Harnessx: A composable, adaptive, and evolvable agent harness foundry. *arXiv preprint arXiv:2606.14249*, 2026.

Lanxiang Hu, Mingjia Huo, Yuxuan Zhang, Haoyang Yu, Eric P Xing, Ion Stoica, Tajana Rosing, Haojian Jin, and Hao Zhang. lmgame-bench: How good are llms at playing games? In *International Conference on Learning Representations*, volume 2026, pp. 81308–81356, 2026.

Wen Jiang, Mingmin Chu, Yimeng Tian, Qianxin Zhang, Haofei Yang, Rui Yang, Yang Liu, Tao Lv, and Fangming Li. Harnessevolve: Learning from reference trajectories for reliable agent self-evolution. *arXiv preprint arXiv:2609.00829*, 2026.

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

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

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, et al. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026.

Yuxing Lu, Yicheng Chen, Shanchan Wu, and Sercan Ö Arık. Procedural graphs: Self-evolving execution structures for llm agents. *arXiv preprint arXiv:2609.09153*, 2026.

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, 2026. URL https://arxiv.org/abs/2601.11868.

Jun Nie, Yonggang Zhang, Jun Song, Qianshu Cai, Dahai Yu, Yike Guo, Xinmei Tian, and Bo Han. Tthe: Test-time harness evolution. *arXiv preprint arXiv:2607.08124*, 2026.

OpenSpace Team@HKUDS. OpenSpace: The skill management layer for AI agents. GitHub repository, 2026. URL https://github.com/HKUDS/OpenSpace. Accessed: 2026-09-18.

Tejal Patwardhan, Rachel Dias, Elizabeth Proehl, Grace Kim, Michele Wang, Olivia Watkins, Simon Fishman, Marwan Aljubeh, Phoebe Thacker, Laurance Fauconnet, et al. Gdpval: Evaluating ai model performance on real-world economically valuable tasks. In *International Conference on Learning Representations*, volume 2026, pp. 24005–24040, 2026.

Prithvi Rajasekaran. Harness design for long-running application development, March 2026. URL https://www.anthropic.com/engineering/harness-design-long-running-apps.

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

Vivek Trivedy. Improving deep agents with harness engineering, February 2026. URL https://www.langchain.com/blog/improving-deep-agents-with-harness-engineering.

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. In *International Conference on Learning Representations*, volume 2026, pp. 80356–80386, 2026.

Yuhao Wu, Jingyuan Zhang, Jiajun Shi, Xinping Lei, Qingshui Gu, Yuxuan Zhang, Zexuan Wang, Chen He, Chen Huang, Maojia Song, et al. Harnessdev: Can llms create and evolve their own agent harness? *arXiv preprint arXiv:2609.01437*, 2026.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

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

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

Yilun Yao, Xinyu Tan, Chao-Hsuan Liu, Yaoming Li, Zhengyang Wang, Wenhan Yu, Zhewen Tan, Yuxuan Tian, Guangxiang Zhao, Lin Sun, et al. Harness-bench: Measuring harness effects across models in realistic agent workflows. *arXiv preprint arXiv:2605.27922*, 2026.

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, et al. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*, 2026.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-harness: Harnesses that improve themselves. *arXiv preprint arXiv:2606.09498*, 2026a.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin gödel machine: open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, volume 2026, pp. 104223–104294, 2026b.

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

Tong Zheng, Xidong Wu, Zheng Zhang, Zhankui He, Chaoyi Zhang, Benjamin Coleman, Ruoqiao Wei, Di Bai, Haolin Liu, Rui Liu, Xue Wang, Yue Zhuan, Wang-Cheng Kang, Renkai Xiang, Heng Huang, Xinwu Cheng, and Yunsong Guo. Dream-rsi: Recursive self-improvement through evolving worlds. Preprint, 2026. URL `https://dream-rsi.com/assets/dream-rsi.pdf`.

A LIMITATIONS

Evolution-induced variance. Online harness evolution introduces an additional source of evaluation variance compared with evaluating agents with static harnesses, as variation in the evolver's updates can affect subsequent agent behavior and such effects can compound over a run. To mitigate the impact of this variance, we repeat each experiment three times and report the average, reducing sensitivity to individual runs.

Indirect feedback signal. The evidence available to the evolver for updating the harness is based on recent execution traces, records of previous updates, main-agent requests, and trigger timing. The evolver therefore proposes updates and reviews its previous changes primarily based on such execution feedback. However, such feedback sometimes can be indirect and sparse, making the quality of harness evolution dependent on the evolver's own capabilities. Future work could investigate how to enrich these intermediate feedback signals without access to the final outcome, enabling more robust harness evolution.

B EVALUATION DETAILS

B.1 AGENT HARNESS VERSIONS

We pin mini to mini-swe-agent version `2.2.8` with the `mini.yaml` agent template (using the benchmark task directly as the instance prompt), use the same pinned mini base for miniFlux, and pin Codex to version `0.147.0`.

B.2 BENCHMARK CONFIGURATION

Tables 4 and 5 summarize the evaluation limits and benchmark-specific execution settings. Limits apply independently to each game, episode, or task; model steps and game actions are distinct. For ARC-AGI-3, the action-budget multiplier scales the human baseline action count for each level: the upstream default of 5 permits up to $\lceil 5h_{g\ell} \rceil$ actions, where $h_{g\ell}$ is the human baseline for that level.

Table 4: Benchmark per-task evaluation limits.

Benchmark	Agent step limit	Game action limit	Wall-clock timeout
ARC-AGI-3	Unlimited	action-budget-multiplier=5 (upstream default)	4 hours
Tetris	Unlimited	2,000 decisions (upstream default)	4 hours
GDPval-Rubric	100 (from GDPval-AA v1)	N/A	4 hours

Table 5: Benchmark-specific interfaces and isolation.

Benchmark	Task interface and custom settings	Isolation
ARC-AGI-3	25 games; text-only integer-grid observations and actions through the `arc3` CLI.	Bubblewrap with direct network access disabled.
Tetris	Seeds 42, 43, and 44; native text observations and the `tetris` CLI.	Docker with direct network access disabled.
GDPval-Rubric	44 compatible tasks from the 50-task subset; reference files are provided in the workspace. Only artifacts listed in `submission.txt` are graded.	Docker.

B.3 Evaluation Metrics

ARC-AGI-3 RHAE. We directly follow the official ARC-AGI-3 Relative Human Action Efficiency (RHAE) metric [1] and summarize its calculation below for clarity. This metric measures per-level action efficiency compared to a human baseline, normalized per game, across all games. Specifically, for one play of game g, let $h_{g\ell}$ and $a_{g\ell}$ denote the human baseline and agent action counts for level ℓ, and let $c_{g\ell}$ indicate completion. The level score is

$$q_{g\ell} = \begin{cases} \min\left\{115,\ 100\,(h_{g\ell}/a_{g\ell})^2\right\}, & c_{g\ell} = 1 \text{ and } a_{g\ell} > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

For a game with L_g levels, levels are weighted by their one-based indices, and the game score is capped by the weighted fraction of levels with positive scores:

$$s_g = \min\left\{ \frac{\sum_{\ell=1}^{L_g} \ell\, q_{g\ell}}{\sum_{\ell=1}^{L_g} \ell},\ 100\frac{\sum_{\ell=1}^{L_g} \ell\, \mathbf{1}[q_{g\ell} > 0]}{\sum_{\ell=1}^{L_g} \ell} \right\}, \qquad \mathrm{RHAE} = \frac{1}{G}\sum_{g=1}^{G} s_g. \tag{2}$$

Uncompleted levels therefore contribute zero while remaining in the denominator. An RHAE score of 100% indicates that the agent completes all evaluated levels and matches or exceeds the human efficiency baseline in aggregate. Scores between 0% and 100% reflect a combination of incomplete level coverage and lower action efficiency relative to humans. A score of 0% indicates that no level receives positive completion credit.

Tetris scoring. We follow the upstream GamingAgent (Hu et al., 2026) Tetris scoring metric without modification and present its calculation below for clarity. The environment awards 1 point for each placed piece and 10 points for each cleared line. The adapter sums these increments over the episode. Let p_{st} and ℓ_{st} denote the numbers of pieces placed and lines cleared during game decision t for seed s. The episode score and the mean score for one evaluation run are

$$S_s = \sum_{t=1}^{T_s} (p_{st} + 10\ell_{st}), \qquad \mathrm{Score} = \frac{1}{3}\sum_{s\in\{42,43,44\}} S_s, \tag{3}$$

where T_s is the number of executed game decisions. A decision can contain a sequence of repeated actions, whose score increments are accumulated together.

GDPval rubric score. We follow OpenAI's public rubrics [2] and score each submission as the percentage of maximum achievable points. To obtain the rubric score, we use gpt-5.6-luna as the judge model to determine whether the gradable artifacts satisfy each rubric criterion, assigning a binary score to each criterion and aggregating these scores into the final grade:

$$\text{Task score (\%)} = 100 \times \frac{\max(0,\ \text{earned positive points} - \text{penalty points})}{\text{sum of all positive rubric weights}}. \tag{4}$$

Specifically, for task i, the judge assigns each rubric item j a binary label y_{ij} indicating whether its criterion is satisfied. Given signed rubric weights w_{ij}, earned positive points are $\sum_{j:w_{ij}>0} w_{ij}y_{ij}$, penalty points are $\sum_{j:w_{ij}<0} |w_{ij}|y_{ij}$, and the maximum achievable positive points are $\sum_{j:w_{ij}>0} w_{ij}$. A task with no positive rubric weights receives zero. For example, earning 80 of 100 possible positive points and incurring 10 penalty points yields a task score of 70%.

Let s_i denote the task-level percentage. The table reports the equally weighted mean across the N compatible task attempts,

$$\mathrm{Score} = \frac{1}{N}\sum_{i=1}^{N} s_i \tag{5}$$

Gradable artifacts retain their scores even if agent execution ends in an error or timeout.

[1] `https://docs.arcprize.org/methodology`.

[2] `https://huggingface.co/datasets/openai/gdpval`

B.4 Token Prices

Table 6 lists the token prices used for the dollar cost estimates in Section 4.3. The same rates apply to main-agent and evolver usage.

Table 6: Token prices used in the cost analysis, in USD per million tokens. Output prices include reasoning tokens.

Model	Uncached input	Cache read	Output
DeepSeek-V4-Flash	$0.14	$0.028	$0.28
GLM-5.2	$1.40	$0.26	$4.40

C Token Cost Details

Figures 7 and 8 supplement the cost analysis in Section 4.3, showing main-agent token use for mini, miniFlux, and Codex on ARC-AGI-3 and Tetris, broken down into cache-read, input, and output tokens, summed over 3 repeat runs.

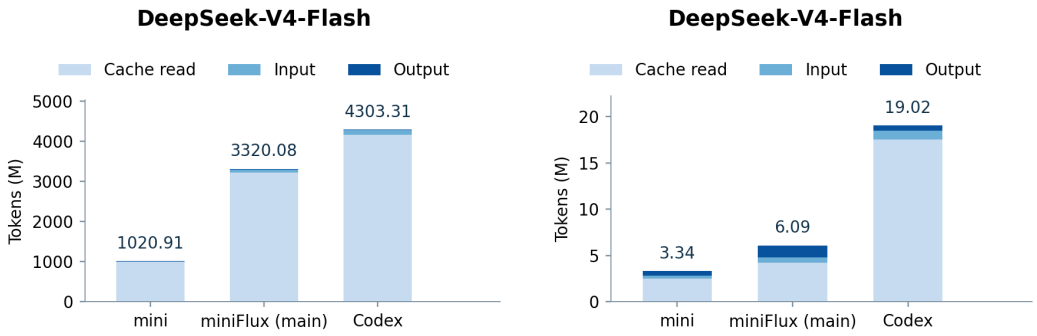


Figure 7: Main-agent token use for DeepSeek-V4-Flash on ARC-AGI-3 (left) and Tetris (right).

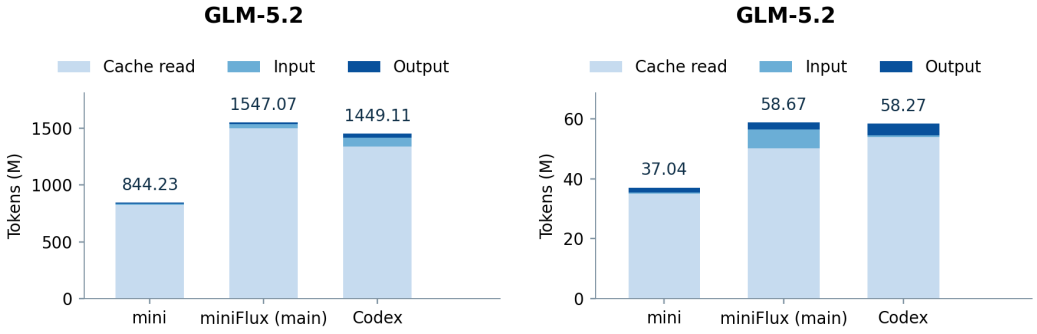


Figure 8: Main-agent token use for GLM-5.2 on ARC-AGI-3 (left) and Tetris (right).

D SEED COMPONENT REPOSITORY

Table 7 lists the default runtime prompt and built-in source components available in the miniFlux seed repository. The minimal profile initially enables only the Bash tool. When agent-requested evolution is enabled, the host additionally installs the protected `harness_update` tool. Other seed components are available for selective installation and adaptation during execution. Runtime prompt guidance is maintained separately in memory, rather than a persistent source component.

Table 7: Seed components in miniFlux. Default indicates whether the component is enabled at task initialization, with agent-requested evolution enabled. † means default installed only when agent-requested evolution is enabled. The runtime hand-off is injected only after a harness update.

Class	Component	Default	Description
Prompt	Runtime prompt guidance	no	Defaults to "Harness updated." The evolver can revise this text, which is delivered with a compact update digest after a harness update.
Tools	`bash`	yes	Execute shell commands in the task environment.
	`harness_update`†	yes	Request an evolution cycle with an optional reason.
	`read_file`	no	Read text files with line numbers and pagination.
	`write_file`	no	Create or overwrite files, with syntax checks where supported.
	`patch`	no	Apply targeted replacements or multi-file patches.
	`search_files`	no	Search file contents or locate files by name.
	`web_search`	no	Search the web and retrieve result summaries.
	`web_extract`	no	Extract content from web pages or PDF URLs.
	`todo`	no	Maintain a task list and track completion status.
Middleware	`budget_reminder`	no	Provide reminders about step, time, and context budgets.
	`message_sanitization`	no	Repair malformed argument JSON and invalid characters.
	`pre_exit_validation`	no	Intercept submission or exit to request reconfirmation.
	`runtime_friction`	no	Report error cascades, budget pressure, and tool timing.
	`safety_blocklist`	no	Block configured commands that could disrupt execution.
	`tool_guardrails`	no	Detect repeated failed or non-progressing calls and provide recovery guidance.
Context	`compressor`	no	Compress accumulated context through summarization, pruning, and truncation.

E EVOLVED COMPONENTS

We present **arc3_analyze()**, a tool created by DeepSeek-V4-Flash during ARC-AGI-3 task re86-8af5384d. The example uses the saved final implementation, including subsequent refinements. It reads the grid cached by the arc3 tool and returns structured geometric summaries without issuing a game action.

Tool description. The following description is reproduced verbatim from the tool registration. References to particular shapes and cell values reflect the evolved tool's task-specific description.

arc3_analyze: tool description

```
Analyze the arc3 grid persisted by the arc3 tool at /tmp/arc3_tool_state.json.
Run `arc3 observe` first so the file reflects the current frame/level.
query='objects' segments a cell value (default 8: the moving shape and the 8
block centers) into connected components with bbox, size, cell list and exact
template coordinates - pass min_size to drop tiny components and/or limit to
keep only the N largest, since a 4-connected value-8 grid can have hundreds of
singleton path cells; the reply always includes total_components, size_buckets
and omitted so you can gauge structure without a huge dump; query='holes' finds
enclosed 0-regions (e.g. the chevron's hole, which acts as its center);
query='diff' returns the persisted cell-change list; query='rows' returns exact
integer rows (e.g. floor row 63); query='summary' returns per-value counts. Use
this instead of hand-parsing grids or writing throwaway python.
```

Tool schema. The input schema below preserves all types, constraints, and required fields; property-level descriptions are omitted for compactness. Only query is required. For objects, the handler defaults to value 8, connectivity 4, min_size 1, and limit 0 (unlimited). The rows field supplies row indices for the rows query.

arc3_analyze: input schema

```
{
  "type": "object",
  "properties": {
    "query": {
      "type": "string",
      "enum": ["summary", "objects", "holes", "diff", "rows"]
    },
    "value": {"type": "integer", "minimum": 0, "maximum": 15},
    "connectivity": {"type": "integer", "enum": [4, 8]},
    "min_size": {"type": "integer", "minimum": 1},
    "limit": {"type": "integer", "minimum": 0},
    "rows": {"type": "array", "items": {"type": "integer"}}
  },
  "required": ["query"]
}
```

Core implementation. The objects query in arc3_analyze converts the cached grid into structured geometric summaries, reducing manual cell-by-cell parsing. Its core function, _components(), uses flood fill to group cells of a selected value under four- or eight-neighbor connectivity. Each region is described by its size, bounding box, absolute cell coordinates, and a shape template relative to the bounding-box origin, allowing shape comparison across positions. Regions are sorted by size for inspection, and coordinates follow (row, column) order. The omitted imports provide deque from collections and Any from typing.

arc3_analyze: core implementation

```
def _dirs(connectivity: int) -> list[tuple[int, int]]:
    if connectivity == 8:
        return [(dr, dc) for dr in (-1, 0, 1) for dc in (-1, 0, 1) if dr or dc]
    return [(1, 0), (-1, 0), (0, 1), (0, -1)]


def _components(
    grid: list[list[int]], value: int, connectivity: int
) -> list[dict[str, Any]]:
    rows, cols = len(grid), len(grid[0])
    dirs = _dirs(connectivity)
    seen = [[False] * cols for _ in range(rows)]
    comps: list[dict[str, Any]] = []
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] != value or seen[r][c]:
                continue
            cells = set()
            q = deque([(r, c)])
            seen[r][c] = True
            while q:
                cr, cc = q.popleft()
                cells.add((cr, cc))
                for dr, dc in dirs:
                    nr, nc = cr + dr, cc + dc
                    if (
                        0 <= nr < rows
                        and 0 <= nc < cols
                        and not seen[nr][nc]
                        and grid[nr][nc] == value
                    ):
                        seen[nr][nc] = True
                        q.append((nr, nc))
            rs = [x[0] for x in cells]
            cs = [x[1] for x in cells]
            minr, maxr = min(rs), max(rs)
            minc, maxc = min(cs), max(cs)
            comps.append(
                {
                    "value": value,
                    "size": len(cells),
                    "bbox": [minr, minc, maxr, maxc],
                    "cells": sorted(cells),
                    "template": sorted((x[0] - minr, x[1] - minc) for x in
    cells),
                }
            )
    comps.sort(key=lambda o: (-o["size"], o["bbox"][0], o["bbox"][1]))
    return comps


def _holes(grid: list[list[int]]) -> list[dict[str, Any]]:
    ...  # Implementation omitted.
```

The arc3_analyze tool handler loads /tmp/arc3_tool_state.json, returning an error if no usable grid is available. For objects, it calls _components(), applies min_size and the output limit, and reports total, size-bucket, and omitted counts. Other queries return cell-value counts (summary), enclosed zero-valued regions (holes), cached changes (diff), or selected valid rows (rows). The holes query calls _holes(), which excludes boundary-reachable zeros using four-neighbor flood fill and reports each enclosed region's cells, size, bounding box, bounding-box midpoint, and adjacent nonzero values.

F AGENT PROMPTS

The boxes below present the miniFlux's main-agent system prompt and evolver system prompts from the implementation. The mini baseline shares the same main-agent system prompt.

F.1 MAIN-AGENT SYSTEM PROMPT

Main-agent system prompt (from `mini.yaml`)

```
You are a helpful assistant that can interact with a computer.
```

F.2 EVOLVER USER PROMPT TEMPLATE

Braced fields are substituted at runtime. The trajectory window contains the latest available main-agent steps up to the configured window size, represented as assistant output, tool calls, and tool results. Previous updates are supplied as compact metadata records. The entire `main_agent_reason` block is included only for an agent-requested cycle; scheduled cycles omit it. The call budget and trigger step are runtime values.

Evolver user prompt template

```
<original_user_prompt>
{original_user_prompt}
</original_user_prompt>

<latest_trajectory_window>
{recent_main_agent_steps_json}
</latest_trajectory_window>

<previous_harness_updates>
{previous_harness_updates_json}
</previous_harness_updates>

<main_agent_reason>
{main_agent_reason}
</main_agent_reason>

<evolution_budget>
At most {max_steps} model calls; triggered after main-agent step {trigger_step}.
</evolution_budget>
```

F.3 EVOLVER SYSTEM PROMPTS

Scheduled cycles use the following system prompt.

Evolver system prompt: scheduled trigger

```
You are MiniEvolver, a meta agent that evolves the harness used by the main
    agent during execution to improve the main agent's performance and
    efficiency. Base every update on concrete patterns in the recent
    trajectory. You may modify the runtime prompt, tools, middleware, and
    context policy. Do not solve the user's task here.

## Workflow
1. Inspect the recent trajectory to identify recurring patterns, failures,
    successes, bottlenecks, and missing capabilities. Distinguish persistent
    issues from isolated events.
2. Call `harness_inspect` with no target to see inspectable target classes and
    installed components. Inspect a component class to list available
    built-ins, then inspect the relevant component source or manifest before
    changing it.
3. Choose the smallest justified mutation path:
   - Runtime prompt: add or revise concise guidance only when the trajectory
    shows that behavioral guidance would help.
   - Active component: patch its `component.py` with `harness_patch`. In replace
    mode, omit `old_string` to replace the whole file.
   - Built-in component: install it unchanged, or patch it before installation
    when adaptation is necessary.
   - Missing capability: use `harness_create` for a `tools.*` or `middleware.*`
    component, inspect and implement the scaffold, then install it. Each tool
    component must expose exactly one tool.
4. Validate the update results. If a previous mutation was wrong, correct it or
    use `harness_rollback`. Remove components only when the trajectory provides
    evidence that they are harmful.
5. Call `finish_evolution` to complete the cycle, including when no change is
    justified.

## Guidelines
Focus on the following aspects when deciding whether to evolve the harness:
- Repeated failures (tool loops, timeouts, missing dependencies, etc.): fix the
    relevant tools, build protective middleware, or add concise prompt guidance
    to prevent recurrence.
- Successful patterns (fast progress, useful discoveries, effective tool use,
    etc.): reinforce them in the prompt or add tools that make those strategies
    easier and more reliable.
- Limitations and inefficiencies (unproductive tool use, missing capabilities,
    lack of progress, etc.): create and install better tools, and add prompt
    guidance that reminds the agent of the updates or let it explore better
    strategies.
- Long-horizon, complex task (more than 40 steps, long output, context growing
    rapidly): install and configure the built-in context-management component,
    add budget reminders, and use delivery-validation middleware.

IMPORTANT: Do not make speculative changes because a component might be
    improved. Mutate only when the trajectory provides clear evidence of a
    concrete problem and the proposed change directly addresses it. If the
    evidence is weak, make no changes and call finish_evolution immediately.
    Repeated or similar updates usually have diminishing returns, so do not
    revisit a component without new evidence. If the trajectory is healthy, you
    do not need to inspect or modify any part of the harness and call
    finish_evolution immediately.
```

Agent-requested cycles replace the opening and first workflow step with the text below.

Evolver system prompt: agent-requested trigger

```
You are MiniEvolver, a meta agent that evolves the harness used by the main
    agent during execution to improve the main agent's performance and
    efficiency. Read the main agent's reason when non-empty and treat it as the
    main agent's diagnosis to validate against concrete evidence in the recent
    trajectory. You may modify the runtime prompt, tools, middleware, and
    context policy. Do not solve the user's task here.

## Workflow
1. Read the main agent's reason when non-empty, then inspect the recent
    trajectory. Use the reason to focus your inspection, but do not assume the
    diagnosis is correct. Identify recurring patterns, failures, successes,
    bottlenecks, and missing capabilities. Distinguish persistent issues from
    isolated events.

...
[The omitted remainder is identical to the scheduled-trigger prompt above.]
```

G Complementary Triggering Modes

This trajectory from DeepSeek-V4-Flash on ARC-AGI-3 task `ls20-9607627b` illustrates an timely intervention enabled by complementary triggering modes (Section 5.2). Scheduled updates create a game wrapper and later add structured grid metrics. When the metrics reveal malformed parsing, the main agent requests a repair at step 46, before the nominal next scheduled deadline at step 60. The excerpts retain the key calls, results, and runtime notices; reasoning, evolver steps, and other interactions are omitted.

Steps 20 and 40: scheduled updates introduce and extend the component. The first update creates and installs `arc3`; the second adds bounding boxes and cell counts. The parser's frame-accumulation bug already exists before the metrics are added.

Steps 20 and 40: component creation and extension

```
[Step 20: runtime update notice; excerpt]
Updated: tools.arc3, prompt.
Harness updated: a new `arc3` tool is installed.
[...]

[Step 40: runtime update notice; excerpt]
Updated: context.compressor, tools.arc3, prompt.
[...]
- The `arc3` tool now returns structured fields every call: `player` (C tile
    bbox: row_min/row_max/col_min/col_max/count), `fuel` (bottom B bar
    bbox/count), `trail_9s`, `budget_used/total/left`, and `changed_cells` vs
    the previous frame. Use these instead of hand-copying grids into bash.
[...]
```

Step 43: the output exposes malformed grid metrics. The wrapper concatenates six animation frames into one board, producing row indices outside the 64-by-64 grid and inflated cell counts. A subsequent observation at step 45 returns the same malformed metrics.

Step 43: malformed metrics

```
[Main-agent tool call]
arc3({"command": "act ACTION1"})

[Tool result; excerpt, returncode = 0]
"budget": "19/110 for level 0",
"frame": "Frame 5:",
[...]
"player": {"row_min": 30, "row_max": 351, "col_min": 19, "col_max": 23, "count":
    60},
"fuel": {"row_min": 61, "row_max": 382, "col_min": 18, "col_max": 54, "count":
    444},
"trail_9s": {"row_min": 11, "row_max": 380, "col_min": 3, "col_max": 37,
    "count": 270},
[...]
```

Step 46: the main agent requests a repair. The main agent reports the impossible coordinates and inflated counts through `harness_update`. Its description of the grid appearing twice is an initial diagnosis; the recorded output contains six stacked frames.

Step 46: agent-requested update

```
[Main-agent tool call]
harness_update({"reason": "The arc3 tool's output is now malformed: the grid
    field is duplicated (appears 2x), and structured fields (player bbox
    row_max=351, fuel count=444, trail count=270) are wildly inflated, making
    it impossible to reliably observe the true 64x64 game state. This started
    after the latest harness/tool update and blocks accurate play."})
```

Steps 46–47: the parser is repaired and single-frame metrics are restored. The repair clears accumulated rows at each new frame header, retaining only the final frame. The next observation reports valid coordinates and corrected counts while the action budget remains at 19/110.

Steps 46–47: repair and corrected observation

```
[Step 46: runtime update notice; excerpt]
Updated: tools.arc3, prompt.
[...]
It now keeps only the final (live) frame, so the structured fields and ASCII
    `grid` are accurate single-frame values. Call `arc3 observe` to get the
    current live state and continue the level from there.

[Step 47: main-agent tool call]
arc3({"command": "observe"})

[Tool result; excerpt, returncode = 0]
"budget": "19/110 for level 0",
[...]
"player": {"row_min": 30, "row_max": 31, "col_min": 19, "col_max": 23, "count":
    10},
"fuel": {"row_min": 61, "row_max": 62, "col_min": 18, "col_max": 54, "count":
    74},
"trail_9s": {"row_min": 11, "row_max": 60, "col_min": 3, "col_max": 37, "count":
    45},
[...]
```