Harness of Harness: Multi-Day Autonomous Software Development with Continual Improvement

Haoyang Yan[1], **Minle Su**[1], **Hangfan Zhang**[1], **Zhanhao Li**[1], **Chen Zhang**[1], **Shao Zhang**[1], **Yang Chen**[1], **Lei Bai**[1], **Shuyue Hu**[1]

[1]Shanghai Artificial Intelligence Laboratory

{*xxx,xxx*}*@pjlab.org.cn*

Abstract

Coding agents powered by large language models (LLMs) have demonstrated remarkable capabilities, yet their practical effectiveness depends heavily on harnesses that coordinate tool use, context management, execution, and the integration of resulting evidence. Here we study a real-world challenge: software development from scratch, where agents must autonomously transform high-level requirements into complete, functional software systems. We propose Harness-of-Harness (HoH), a framework that builds upon existing coding-agent harnesses and empowers them with iterative self-improvement capabilities. HoH enables a continuous planning–coding–testing cycle, where agents iteratively generate development plans, refine software artifacts, and incorporate execution evidence to improve both the artifacts and subsequent development decisions. We evaluate HoH on GameCraft-Bench, FrontierSWE, and ProgramBench using three harness–model configurations: Codex with GPT-5.5, OpenCode with DeepSeek-V4-Pro, and Pi with MiniMax-M3. Across these settings, HoH consistently improves over standalone harnesses. Further analysis shows that HoH continues to benefit from additional iterations and maintains its advantage under comparable token budgets.

Keywords: Coding Agents, Software Engineering, Iterative Development, Agent Harnesses

Code: Flesymeb/GameLoop

Project Page: flesymeb.github.io/GameLoop

1 Introduction

Coding agents have emerged as a prominent application for recent large language models (LLMs). However, their practical effectiveness depends not only on the underlying model but also on the harness: the operational framework that transforms a general-purpose LLM into an effective coding agent [11, 13, 22]. Modern coding-agent systems such as Codex, OpenCode, and Claude Code rely on harnesses to govern how a model invokes tools, manages context and memory, controls execution, interprets feedback, and orchestrates subagents.

Although the term harness has recently gained prominence [9, 12, 26, 34], coding agents have been an active area of research for years. Earlier coding agents primarily focused on localized assistance, including code generation, function completion, and answering programming-related queries [3, 6]. More recent coding agents have evolved to address increasingly complex tasks, such as navigating large-scale codebases, and resolving repository-level issues that span multiple files and components [7, 24, 29, 31]. However, these tasks still provide agents with an existing codebase, thereby allowing them to operate within a well-defined environment with clear objectives and constraints.

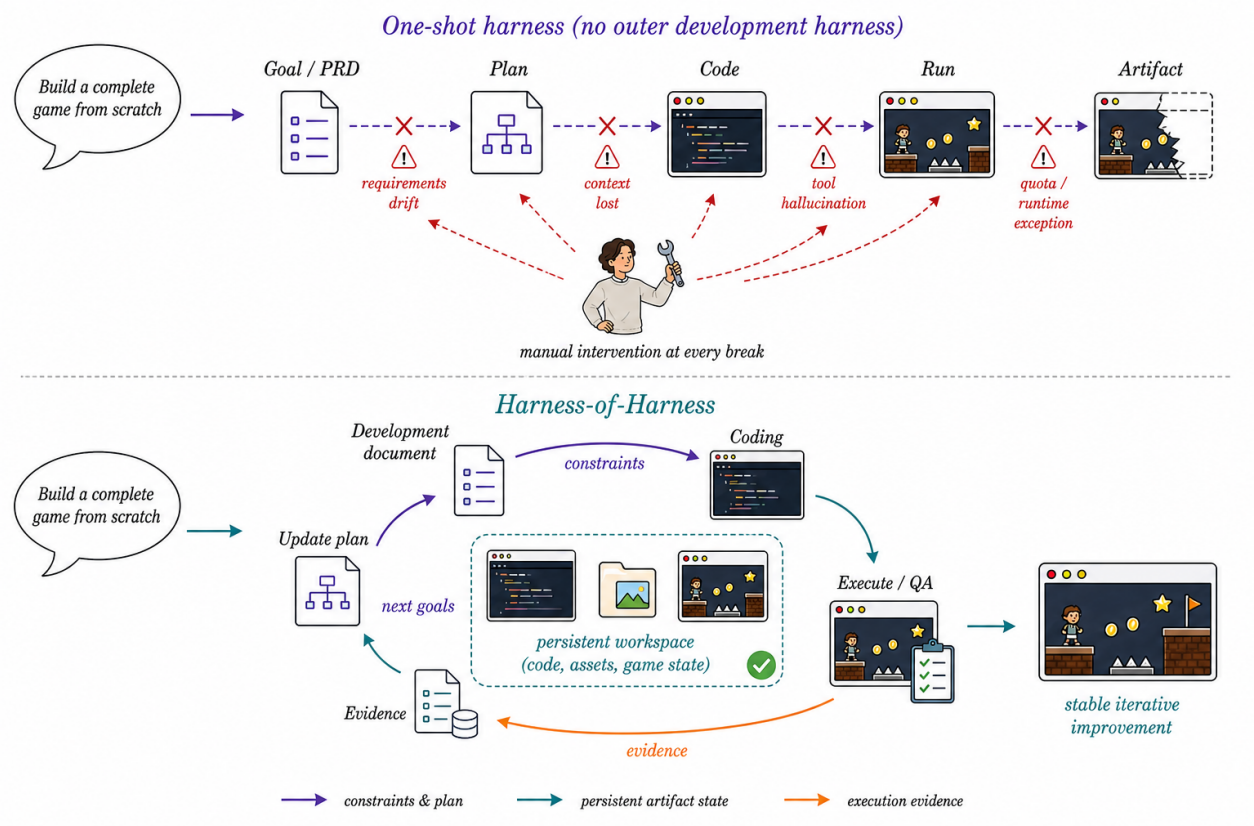

Figure 1 | From manually maintained harnesses to Harness-of-Harness. A conventional coding-agent harness is typically invoked as a linear development pass. When requirement drift, context loss, tool hallucination, or runtime exceptions occur, a human developer must inspect the break, preserve useful state, repair the next prompt, and reinvoke the harness. HoH replaces this manual maintenance with an outer harness that supervises the underlying coding harness across iterations: it preserves the artifact workspace, converts execution evidence into updated development instructions, and drives the next planning–coding–testing cycle.

In this paper, we investigate software development *from scratch*, a critical next step for coding agents toward real-world software engineering. Unlike tasks that begin with an existing codebase and established context [7, 19], this setting requires agents to autonomously transform high-level and often incomplete requirements into complete and functional software systems. Also known as *greenfield* software development, this paradigm represents a broader and more open-ended form of software development, requiring effective support for long-horizon planning, coordination, execution, and adaptation throughout the entire software development lifecycle. This motivates an important yet underexplored research question: how can coding-agent harnesses be leveraged to support software development from scratch?

Figure 1 illustrates this shift from manually maintained harness use to HoH, where an outer harness replaces human intervention by maintaining the development state, evidence, and next-step instructions for the underlying coding harness.

To this end, we introduce Harness-of-Harness (HoH), a new approach that extends existing coding-agent harnesses to support greenfield software development. Rather than designing another standalone harness, HoH builds upon existing harnesses and empowers them with iterative self-improvement capabilities through a continuous planning–coding–testing cycle. The iterative design of HoH is motivated by the observation that real-world software development is inherently an

incremental process, where developers continuously refine development plans, implement changes, and validate outcomes throughout the development lifecycle. Following this principle, each HoH iteration consists of three stages: planning, which transforms requirements and execution evidence from preceding artifacts into actionable development plans; coding, which implements and integrates new components while refining existing artifacts; and testing, which evaluates the artifact from multiple perspectives, including functionality, usability, and aesthetic quality. The resulting execution evidence, together with the updated artifact, is carried forward to subsequent iterations, forming a self-improving loop that progressively enhances not only the software artifacts but also subsequent development decisions.

In our experiments, we evaluate HoH on GameCraft-Bench, FrontierSWE, and ProgramBench, which respectively target game development, long-horizon repository engineering, and cleanroom program reconstruction in greenfield settings. We consider three distinct harness–model configurations: Codex with GPT-5.5 (high), OpenCode with DeepSeek-V4-Pro, and Pi with MiniMax-M3. Across the three benchmarks, HoH consistently improves performance through iterative development, achieving substantial gains over the corresponding standalone harnesses. For example, on GameCraft-Bench, three iterations of HoH improve average scores by 16.62–22.08 absolute points across the three configurations; on FrontierSWE, HoH similarly improves rewards by 0.07–0.29; on ProgramBench, HoH improves *Avg. Test Pass Rate* by 6.09–16.85 points. On GameCraft-Bench and FrontierSWE, performance also improves consistently over successive development cycles. In a pass-controlled comparison on GameCraft-Bench, HoH outperforms repeated Vanilla continuation. Qualitative analyses on GameCraft-Bench show that HoH supports long-horizon software refinement, enabling agents to continuously expand functionality, enrich gameplay mechanics, improve visual presentation, and produce increasingly sophisticated interactions over iterations.

Our key contributions are summarized as follows:

- We formulate a new problem of leveraging existing coding-agent harnesses for software development from scratch, going beyond local code generation and repository-level modification.
- We introduce *Harness-of-Harness*, an iterative approach that enables coding agents to progressively improve software artifacts and overall development strategies through continuous planning–coding–testing cycles.
- Extensive evaluations across multiple harness-model configurations show that HoH consistently improves performance over standalone harnesses, benefits from additional iterations, and enables significant improvements in software quality across multiple dimensions.

2 Related Work

Agent Harnesses. The practical capability of an LLM agent depends not only on the underlying model but also on the harness that governs how the model receives information, acts in an environment, and responds to execution feedback. Existing research improves agent performance through the harness by integrating and optimizing complementary components, including model-facing prompts and context, memory, tools and reusable skills, execution control, verification, and agent orchestration [11, 13, 15, 25, 33, 35, 38]. These components provide the model with task-relevant state, translate model outputs into executable actions, and return execution outcomes for subsequent correction, thereby improving task performance and reliability. Recent work, including AutoHarness, Meta-Harness, and Self-Harness, further automates this process by treating the harness itself as an object of synthesis, search, or iterative refinement [9, 16, 34]. These studies improve agent performance by optimizing

the harness itself. HoH targets a different level of optimization: the evolving software project. HoH builds on existing agent harnesses and iteratively improves an evolving software project through repeated implementation, evaluation, and refinement.

Agentic Systems for Software Development. Research on coding agents has expanded from localized assistance toward increasingly complete forms of software development. This progression spans three broad task settings. **(i) Localized Programming.** Early work focused on generating or refining functions and self-contained programs from explicit specifications and tests [3, 6]. **(ii) Repository-Level Issue Resolution.** As task scope expanded, subsequent work moved from isolated programs to resolving issues within existing repositories [7, 24, 29, 30, 31]. SWE-bench formalized this setting using real-world repository issues, while SWE-agent provided a representative agent–computer interface for inspecting, editing, and testing repository code. **(iii) Greenfield Software Development.** More recently, research has moved beyond bounded issue resolution toward sustained repository evolution, open-ended engineering objectives, and greenfield project construction [4, 5, 8, 14, 17, 18, 21, 23, 32, 36]. FrontierSWE includes from-scratch implementation tasks that require agents to build complex systems under open-ended functional and performance objectives, while GameCraft-Bench evaluates the construction of complete, playable games from natural-language specifications. Greenfield software development requires agents to build and refine a coherent software artifact over an extended development trajectory. Existing coding harnesses typically organize development within a bounded episode, providing limited support for preserving project decisions, verified functionality, and evaluation evidence across subsequent revisions. HoH builds on these harnesses and extends their use to iterative greenfield development by maintaining continuity across planning, implementation, and evaluation cycles.

3 Method

In this paper, we introduce Harness-of-Harness (HoH), a framework that organizes planning, coding, and testing into an iterative loop for end-to-end software engineering. The loop operates on a shared project workspace that evolves across iterations. Figure 2 illustrates the overall framework.

3.1 Problem Formulation

We use the term coding harness to refer to the scaffolding through which a language model interacts with a software environment. Let H denote the harness–model configuration used for development. It comprises the harness implementation, underlying model, role-specific model-facing instructions, available tools and runtime policy. These components remain fixed across iterations within a run; HoH changes the artifact and iteration-specific context supplied to H, rather than the harness itself. For a software-development task, let $\mathcal{S}$ denote the public task specification, including a product requirements document (PRD) and any publicly available auxiliary materials. Let A_0 denote the initial software artifact—the complete project workspace, including source code, configuration, and task-specific resources. It may be an empty or scaffolded workspace or an existing repository. Given the public task specification $\mathcal{S}$, the initial artifact A_0, the fixed harness–model configuration H, and an iteration budget $T \geq 1$, the goal of HoH is to produce a final artifact A_T that satisfies $\mathcal{S}$:

$$\mathrm{HoH}[H] : (\mathcal{S}, A_0, T) \longmapsto A_T. \tag{1}$$

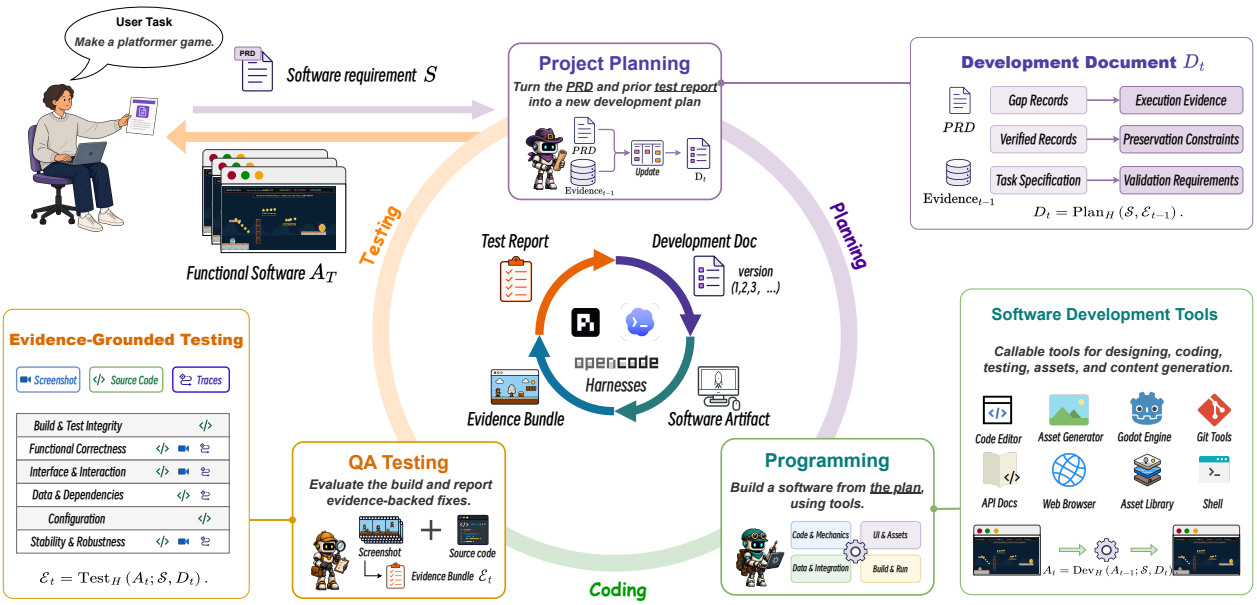

Figure 2 | Overview of Harness-of-Harness (HoH). In each iteration, the *Project Planner* produces an iteration-specific development document, the *Developer* updates the software artifact in the shared project workspace, and the *QA Tester* collects execution evidence. The updated artifact remains in the workspace, while the evidence informs planning in the next iteration.

3.2 Harness-of-Harness Design

HoH coordinates three roles through an iteration-specific development document. Each role is instantiated through a separate invocation of H, and all three invocations operate on the same evolving project workspace under role-specific policies: the *Project Planner* produces plans without modifying the artifact, the *Developer* may modify the workspace, and the *Quality Assurance (QA) Tester* may inspect and execute it but remains read-only. Within these policies, the roles use software-development tools for project version management, asset and implementation-reference retrieval or generation, project execution, runtime debugging and testing, and execution-evidence collection. At iteration t, the *Project Planner* produces the development document D_t. The *Developer* uses D_t to continue development from A_{π_t}, the previously recorded artifact selected as the starting point for iteration t, and produces the candidate artifact A_t in the shared project workspace. The *QA Tester* executes and inspects A_t against the validation requirements in D_t and produces the evidence bundle $\mathcal{E}_t$. The document D_t specifies the current development targets, existing functionality to preserve, unresolved issues, and validation requirements. Let $C_t = \mathrm{Claims}(\mathcal{S}, D_t)$ denote the checkable claims derived from the public task specification and the development document. To retain the connection between each QA finding and its observable basis, we represent the evidence bundle as a set of claim–evidence records:

$$\mathcal{E}_t = \{(c_i, r_i, s_i) \mid c_i \in C_t\}, \quad s_i \in \{\text{verified}, \text{gap}\}. \tag{2}$$

Here, c_i is a checkable claim, r_i contains the observable execution records available during development for that claim—such as build and test outcomes, runtime traces, screenshots, replays, or asset inspections—and s_i records the resulting QA status. Records with s_i = verified form the verified-record subset $\mathcal{E}_t^{\text{ver}}$, whereas those with s_i = gap form the gap-record subset $\mathcal{E}_t^{\text{gap}}$.

Closed-Loop Dynamics. HoH carries two forms of state across iterations: the evolving software artifact A_t and the evidence bundle $\mathcal{E}_t$. With $\mathcal{S}$ and H fixed throughout a run and $\mathcal{E}_0 = \emptyset$, the framework selects A_{π_t} from the recorded candidate lineage as the starting artifact for iteration t, where $\pi_t \leq t-1$. Normally, $\pi_t = t-1$; if the latest candidate is host-latched as regressed or unusable, π_t may refer to an earlier recorded candidate. A complete iteration is summarized as

$$\left(A_{\pi_t}, \mathcal{E}_{t-1}\right) \longmapsto \left(D_t, A_t, \mathcal{E}_t\right), \qquad t = 1, \ldots, T. \tag{3}$$

Within iteration t, D_t serves as the shared specification for development and testing. The artifact A_t carries the current implementation into the next development pass, while $\mathcal{E}_t$ feeds execution observations and QA findings into the next planning stage. Together, these two state channels drive the loop from one iteration to the next.

The public task specification $\mathcal{S}$, including the PRD or equivalent public requirements, remains the global reference throughout the run. At each iteration, the *Project Planner* uses gap records to set development priorities and verified records to identify functionality that should be preserved. It records both in D_t, together with validation requirements derived from $\mathcal{S}$. The *Developer* and *QA Tester* use D_t as a shared reference, aligning implementation and verification within the iteration. Evidence may change the priorities recorded in the next development document, but it does not replace $\mathcal{S}$. The loop can therefore adapt to observed execution outcomes while remaining aligned with the original requirements.

Planning for the Current Iteration. At the beginning of each iteration t, the *Project Planner* uses the public task specification $\mathcal{S}$ and the preceding evidence bundle $\mathcal{E}_{t-1}$ to produce the development document D_t:

$$D_t = \mathrm{Plan}_H\left(\mathcal{S}, \mathcal{E}_{t-1}\right). \tag{4}$$

The development document contains a prioritized list of update targets, preservation constraints for previously verified functionality, and validation requirements for the current iteration. Since the initial evidence bundle is empty, the first document is derived from $\mathcal{S}$ alone. In subsequent iterations, records in the gap-record subset $\mathcal{E}_{t-1}^{\mathrm{gap}}$ inform the update targets and follow-up validation requirements, whereas records in the verified-record subset $\mathcal{E}_{t-1}^{\mathrm{ver}}$ identify functionality to be covered by the preservation constraints. The *Project Planner* reconciles these records with $\mathcal{S}$ to produce D_t.

Development from the Current Artifact. The *Developer* invokes the fixed harness–model configuration H with the public task specification, the development document for the current iteration, and A_{π_t}, the artifact selected as the starting point for the current iteration:

$$A_t = \mathrm{Dev}_H\left(A_{\pi_t}; \mathcal{S}, D_t\right). \tag{5}$$

Development proceeds in the project workspace represented by A_{π_t}, where the existing source code, configuration, assets, and project resources remain directly available. The ordered update targets in D_t place build blockers and evidence-backed defects ahead of unmet requirements and capability extensions; after higher-priority targets are addressed, the remaining development budget can be allocated to extensions. The preservation constraints in D_t remain active across these changes, making functionality supported by preceding evidence an explicit condition on subsequent development. All update targets in D_t are presented to one invocation of H, which retains control over tool use and local implementation decisions while operating under the shared document. The resulting code, assets, and project metadata are written back to the workspace, yielding A_t for subsequent testing.

Algorithm 1 Harness-of-Harness

Input: public specification $\mathcal{S}$, initial artifact A_0
Parameter: fixed harness–model configuration H, iteration budget T
Output: final artifact A_T

$\mathcal{E}_0 \leftarrow \emptyset$
for $t = 1, \ldots, T$ **do** ▷ *HoH iteration*
  $D_t \leftarrow \text{Plan}_H(\mathcal{S}, \mathcal{E}_{t-1})$
  $A_t \leftarrow \text{Dev}_H(A_{\pi_t}; \mathcal{S}, D_t)$
  $C_t \leftarrow \text{Claims}(\mathcal{S}, D_t)$
  $\mathcal{E}_t \leftarrow \emptyset$
  for $c_i \in C_t$ **do** ▷ *Claim-wise verification*
    $r_i \leftarrow \text{Observe}(A_t, c_i)$
    $s_i \leftarrow \text{Assess}(c_i, r_i)$
    $\mathcal{E}_t \leftarrow \mathcal{E}_t \cup \{(c_i, r_i, s_i)\}$
  end for
end for
return A_T

Evidence-Grounded Testing. The *QA Tester* invokes the fixed harness–model configuration H to execute and inspect the updated artifact A_t. The checkable claims in C_t are derived from the public requirements, the validation requirements in D_t, and the functionality covered by its preservation constraints. Testing produces the evidence bundle defined in Equation 2:

$$\mathcal{E}_t = \text{Test}_H\left(A_t; \mathcal{S}, D_t\right). \tag{6}$$

The testing operator is implemented through claim-wise observation and assessment. For each claim c_i in C_t, $\text{Observe}(A_t, c_i)$ collects the relevant observable execution records r_i, and $\text{Assess}(c_i, r_i)$ assigns the status s_i. The corresponding claim–evidence record is assigned verified and included in $\mathcal{E}_t^{\text{ver}}$ only when the cited execution records provide sufficient observable support for the claim. Records associated with observed failures, unmet requirements, regression risks, or insufficient evidence are assigned gap and included in $\mathcal{E}_t^{\text{gap}}$. Verified records identify functionality that the *Project Planner* can carry forward as preservation constraints. Gap records describe unresolved issues and any additional evidence needed for subsequent verification. Both subsets inform the next planning stage and exclude benchmark scores, hidden tests, and evaluator-only feedback.

Overall Procedure. Algorithm 1 summarizes how artifact continuity and execution evidence connect the three roles. Gap records from one iteration inform development targets for the next, while verified functionality is translated into preservation constraints. The final artifact is returned after the prescribed number of iterations.

3.3 Building for Days, Not Turns: Practical Design Lessons

Software lives longer than an agent episode. Greenfield development from a high-level product specification requires repeated planning, implementation, validation, and recovery before a usable artifact emerges, while useful systems may continue to evolve for years as requirements, environments, and user needs change [1, 10]. By contrast, prominent coding-agent benchmarks such as SWE-bench, together with harnesses evaluated in this setting, formulate development as a bounded episode that starts from an established repository and terminates after addressing a single issue [7, 31].

FeatureBench further reports limited success when development spans multiple commits and pull requests [37]. In our multi-day HoH runs, we observed progress stalls associated with loss of relevant context, requirement drift, tool and runtime failures, premature issue closure, and regressions that invalidated later work. Drawing on these observations and the design of HoH, we distill practical lessons for sustaining autonomous development and supporting iterative improvement of the software artifact over extended horizons.

Persistent Project Records Are as Essential as Code for Long-Running Development. Source code records what a software system currently is, but not the full set of intentions, decisions, and evidence needed to determine what it should become next. This distinction becomes critical in long-horizon development. Because current coding agents and harnesses operate with bounded context and imperfect project memory, retaining only the latest codebase forces each invocation to reconstruct the project state from the repository. That reconstruction is inherently lossy: requirements may be reinterpreted, implemented capabilities overlooked, unresolved defects forgotten, and unverified fixes accepted as complete. Development documents, issue histories, and test records should therefore evolve alongside the software artifact rather than disappear with the interaction that produced them. HoH operationalizes this principle by carrying objectives, preservation constraints, verified behavior, unresolved gaps, and artifact lineage across iterations. The codebase and its accompanying records thus constitute a single evolving project state, enabling subsequent invocations to inherit both the implementation and the knowledge required to extend it coherently.

Long-Horizon Planning Should Balance Fixing Existing Problems with Developing New Features. Long-horizon development must pursue two objectives, neither of which is sufficient on its own: stabilizing the current artifact and expanding its capabilities toward the intended system [36, 37]. If development is driven only by observed failures, an agent may remain trapped in local repairs while important capabilities remain unimplemented. Conversely, if development focuses only on capability growth, unresolved blockers and regressions may be buried beneath an increasingly complex artifact. The balance should therefore not be fixed, but adjusted to the current state of the artifact. Before each development pass, HoH reconciles the global specification with the latest execution evidence, making this trade-off an explicit planning decision. Evidence-backed failures determine what should be repaired first, verified behavior defines what subsequent changes must preserve, and unmet requirements indicate where the artifact should grow. The resulting development document orders these demands into iteration-specific priorities, allowing each pass to improve the current artifact while continuing to advance toward the complete system.

Long-Horizon Development Needs Stable Roles, Not Ever-Expanding Agent Teams. Reliable progress over multiple iterations requires implementation and acceptance to remain separate, stable responsibilities. Developer self-checks support rapid iteration but are usually confined to the current change. A target feature may pass its tests even though broader requirements remain unmet, the modification has broken related functionality, or the end-to-end workflow no longer works [20, 27]. Independent acceptance prevents these local checks from becoming the sole basis for declaring completion. Role decomposition should be determined by independent decision boundaries rather than the number of subtasks. A new role is justified only when it contributes distinct authority and a separately verifiable judgment; otherwise, additional handoffs fragment context and accountability without improving reliability [2].

A Harness Should Keep Agents on Track without Holding Them Back. A harness for long-horizon development should constrain delivery without prescribing execution. At the boundary

between invocations, it should define how work is handed off, which outcomes may enter the shared project state, and how incomplete or unstable outcomes are rejected or recovered. These constraints prevent local failures from propagating across iterations. Within an invocation, however, the underlying harness or agent should retain control over its reasoning, tool use, debugging strategy, and implementation path. Such decisions depend on local evidence that cannot be fully anticipated at the outer boundary. Preserving this autonomy allows the underlying system to adapt its behavior, select effective actions, and make full use of its capabilities. Extending delivery constraints into internal execution instead narrows the solution space and makes the system brittle. Effective control is therefore strict about what is delivered, but flexible about how it is produced.

Context Should Persist across Iterations; Knowledge Should Transfer across Projects. Project records, including product requirements, development documents, issue histories, test reports, and execution logs, are long-lived artifacts that evolve and accumulate alongside the code. Successive coding-agent invocations use them to reconstruct goals, current state, prior decisions, unresolved issues, and verified behavior. As this history outgrows an agent's working context, placing the full record in its prompt introduces stale, duplicated, or conflicting information. The agent may then form an outdated project model and drift from current requirements. Records should therefore remain synchronized and versioned with the artifact, preserved durably, and retrieved selectively to expose current and relevant evidence. Experience accumulated across different project and task types also reveals recurring testing procedures, debugging heuristics, tool-use conventions, domain workflows, and failure patterns. Rather than carrying their project-specific details forward, these experiences should be abstracted into reusable knowledge units, such as skills or policies. This distillation converts episodic experience into cumulative capability that future agents can retrieve and apply in other projects. Project-specific context should remain linked to an artifact across iterations, whereas reusable knowledge should accumulate and transfer across projects.

4 Experiments

We evaluate HoH on three software-development benchmarks and three harness–model configurations, comparing final artifact quality with the corresponding Vanilla baselines.

4.1 Experimental Setup

Benchmarks. We evaluate HoH on three benchmarks: GameCraft-Bench [18], FrontierSWE [4], and ProgramBench [32]. GameCraft-Bench comprises 140 tasks across 15 game families, each requiring an agent to construct a complete, playable Godot project from a natural-language specification. We sample 45 tasks using stratified random sampling by game family, selecting three tasks from each of the 15 families with a fixed random seed. For coarse-grained analysis, we additionally organize the 15 families into five broader groups defined in this work: Action, Timing, Strategy, Simulation, and Adventure. The complete sampled task list and our family-to-group mapping are provided in the supplementary material. Due to computational resource constraints, we select 15 tasks from FrontierSWE's 17 tasks for evaluation, comprising 4 Implementation (Impl.), 9 Performance (Perf.), and 2 Research tasks. ProgramBench is a cleanroom program-reconstruction benchmark in which agents receive only a compiled executable and documentation and must rebuild a codebase whose behavior matches the reference program. More details are provided in the supplementary material.

Setting	GameCraft-Bench						FrontierSWE				ProgramBench
	Action	Timing	Strat.	Sim.	Adv.	***Overall***	Impl.	Perf.	Research	***Dominance***	***Pass Rate***†
Codex + GPT-5.5 (high)											
Vanilla	48.74	48.80	44.06	53.63	52.68	49.58	0.21	0.17	1.15	46%	60.41
HoH@1	59.73	53.79	57.01	66.75	61.24	59.71	0.22	0.42	0.76	45%	65.42
HoH@2	64.34	62.03	59.97	71.77	66.11	64.84	0.29	**0.47**	0.59	49%	65.79
HoH@3	**71.02**	**70.26**	**66.13**	**78.42**	**71.76**	**71.52**	**0.30**	**0.47**	**1.16**	**67%**	**66.50**
OpenCode + DeepSeek-V4-Pro											
Vanilla	26.21	24.05	21.27	37.12	25.84	26.90	0.08	0.23	0.57	26%	45.27
HoH@1	27.75	27.40	21.73	43.75	22.44	28.61	0.09	0.23	0.78	34%	55.33
HoH@2	43.22	36.56	33.51	52.26	36.05	40.32	0.10	0.27	0.78	47%	55.66
HoH@3	**49.00**	**45.05**	**43.34**	**55.86**	**51.64**	**48.98**	**0.15**	**0.27**	**0.78**	**49%**	**57.56**
Pi + MiniMax-M3											
Vanilla	45.64	38.20	34.53	48.19	44.26	42.16	0.06	0.12	1.30	36%	35.83
HoH@1	50.59	52.23	38.60	54.00	49.89	49.06	0.10	0.42	**1.89**	65%	48.68
HoH@2	54.70	56.52	42.33	63.20	58.47	55.04	0.11	0.43	**1.89**	**68%**	**53.57**
HoH@3	**58.24**	**62.10**	**44.86**	**64.25**	**64.44**	**58.78**	**0.11**	**0.45**	1.88	66%	52.68

Table 1 | Main results on GameCraft-Bench, FrontierSWE, and ProgramBench. Each harness is evaluated under Vanilla and HoH@1–3. Within each harness and metric, bold values mark the best setting and underlined values mark the second-best; rankings use unrounded values, and exact ties share the same formatting. Bold italic labels distinguish aggregate metrics from task categories. Strat., Sim., Adv., Impl., and Perf. denote Strategy, Simulation, Adventure, Implementation, and Performance, respectively. FrontierSWE reports category scores and Dominance. For ProgramBench, *Pass Rate*† denotes the benchmark's *Avg. Test Pass Rate*.

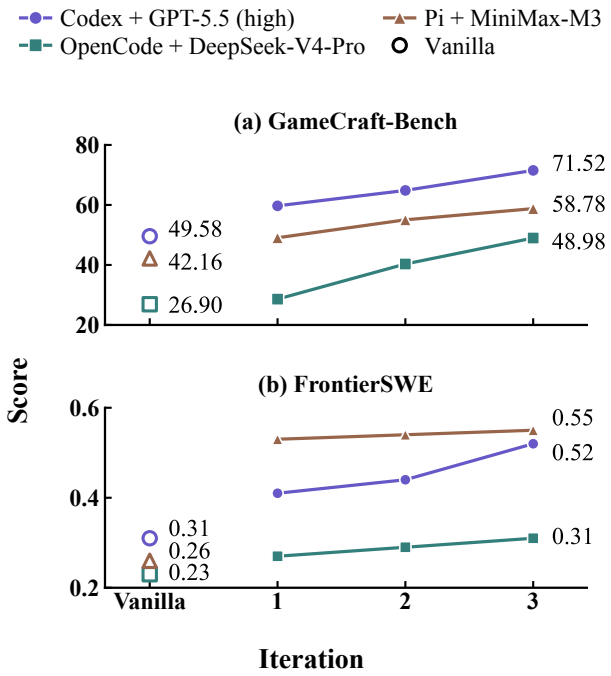

Figure 3 | Overall performance across three iterations on GameCraft-Bench and FrontierSWE, with Vanilla shown for reference.

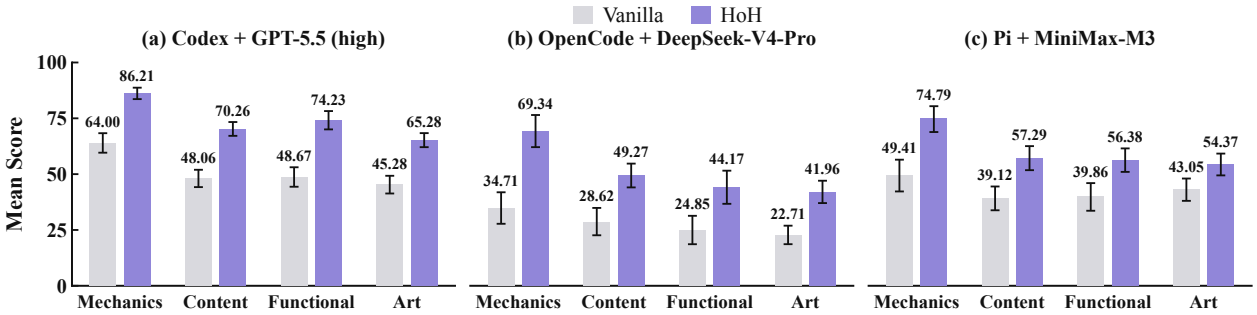

Figure 4 | GameCraft-Bench component scores for Vanilla and HoH ($T = 3$). Results are reported mean score for Core Mechanics, Content Depth, Functional Visuals, and Art and Presentation with 95% bootstrap confidence intervals.

Harnesses and Models. We evaluate HoH with three harness–model configurations: Codex CLI[1] with GPT-5.5 at high reasoning effort, OpenCode[2] with DeepSeek-V4-Pro, and Pi Coding Agent[3] with MiniMax-M3.

Baseline and Evaluation Protocol. We compare HoH against *Vanilla*, the corresponding harness–model configuration without the HoH protocol. Vanilla performs one standard development pass, whereas *HoH@T* performs T planning–coding–testing iterations, with the software artifact and execution evidence carried across iterations; the main experiments use $T = 3$. Intermediate and final HoH artifacts are evaluated only after the complete run, and evaluator outputs are not returned to the development loop. For each task, Vanilla and HoH use identical benchmark-provided initial states and the same underlying harness–model configuration, differing only in the application of the HoH protocol.

Metrics. For GameCraft-Bench, we report the benchmark's *Overall* score. Under this metric, game artifacts that fail to compile or run receive a score of zero, while runnable artifacts are scored by combining Core Mechanics, Content Depth, Functional Visuals, and Art and Presentation using the benchmark-defined weights. We average task-level scores over the 45 tasks and use the benchmark's 0–100 scale. For FrontierSWE, task-specific verifiers assign official rewards, and we report the mean reward over the 15 evaluated tasks. We additionally report the official dominance score, defined as the average task-level win rate against a randomly selected competing configuration from the 12 evaluated harness–condition combinations. For ProgramBench, we report *Avg. Test Pass Rate*, computed as the mean across tasks of the fraction of hidden behavioral tests passed for each task. We use this continuous signal for relative comparisons between Vanilla and HoH and abbreviate it as *Pass Rate*† in Table 1. As a proxy for model-interaction volume, we report provider-reported cumulative input and output tokens from coding-harness model calls, excluding benchmark evaluation. Input totals may include cached context reads; because cache accounting differs across providers, we use these values for within-configuration comparisons rather than direct cross-provider cost comparisons.

4.2 Results and Analysis

HoH improves artifact quality across benchmarks and task categories. Table 1 reports Vanilla and all three HoH iterations for each harness–model configuration. HoH@3 outperforms Vanilla

[1] https://github.com/openai/codex; version 0.142.5.
[2] https://github.com/anomalyco/opencode; version 1.14.30.
[3] https://github.com/earendil-works/pi; version 0.80.10.

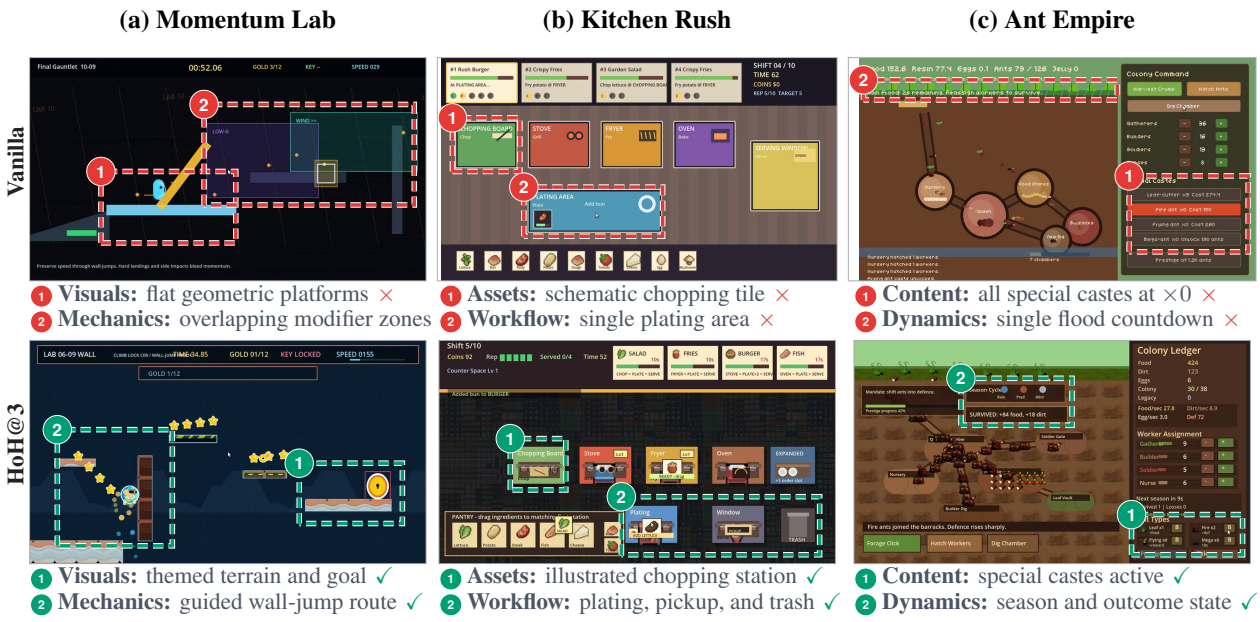

Figure 5 | Qualitative comparison of final game artifacts produced by Vanilla and HoH@3 using Codex with GPT-5.5 (high). Columns show three GameCraft-Bench tasks from distinct game families: Momentum Lab (momentum-based platformer), Kitchen Rush (restaurant-management simulation), and Ant Empire (idle colony-management game). Rows show gameplay frames from Vanilla (top) and HoH@3 (bottom). Numbered dashed boxes identify the regions discussed in the annotations; red crosses and green checks denote limitations and implemented functionality, respectively.

across the three benchmarks under all three configurations. On GameCraft-Bench, mean Overall scores increase from 49.58 to 71.52 for Codex, from 26.90 to 48.98 for OpenCode, and from 42.16 to 58.78 for Pi. On FrontierSWE, rewards increase from 0.31 to 0.52, from 0.23 to 0.31, and from 0.26 to 0.55, respectively. On ProgramBench, *Avg. Test Pass Rate* increases from 60.41 to 66.50 for Codex, from 45.27 to 57.56 for OpenCode, and from 35.83 to 52.68 for Pi. HoH@3 also outperforms Vanilla in every reported task category across the three benchmarks under all three configurations.

HoH improves performance across all evaluated harness–model configurations. The gains are not limited to configurations with a particular level of Vanilla performance. Codex with GPT-5.5 (high), the strongest Vanilla configuration, reaches the highest final GameCraft-Bench score of 71.52 after improving by 21.93 points. Pi with MiniMax-M3 records the largest gain on FrontierSWE, increasing by 0.29 from 0.26 to 0.55, and the largest ProgramBench gain, increasing the average test pass rate by 16.85 points. OpenCode with DeepSeek-V4-Pro starts from the lowest Vanilla score on GameCraft-Bench and FrontierSWE, yet HoH@3 raises its scores to 48.98 and 0.31, respectively, while increasing its ProgramBench average test pass rate from 45.27 to 57.56. OpenCode with HoH@3 further exceeds Codex Vanilla in Action and Simulation on GameCraft-Bench and in Performance on FrontierSWE. Thus, HoH improves configurations that begin at substantially different levels of Vanilla performance.

HoH improves game quality across all evaluation dimensions. Figure 4 reports the four benchmark-defined GameCraft-Bench quality components separately. HoH@3 improves all four components under every harness–model configuration, with gains of 20.00–25.56 points for Codex, 19.25–34.63 points for OpenCode, and 11.32–25.38 points for Pi. For Codex, Functional Visuals shows the largest increase, from 48.67 to 74.23, while Art and Presentation rises from 45.28 to 65.28. The

improvements therefore span gameplay mechanics, content richness, visual clarity, and presentation quality.

Performance continues to improve over successive HoH iterations. Figure 3 shows that mean performance increases monotonically from HoH@1 to HoH@3 across all six combinations of benchmark and harness–model configuration. Beyond the first HoH iteration, the two subsequent iterations improve GameCraft-Bench scores by 9.72–20.37 points and FrontierSWE rewards by 0.02–0.11. This progression is particularly evident for OpenCode on GameCraft-Bench: its advantage over Vanilla grows from only 1.71 points at HoH@1 to 13.42 points at HoH@2 and 22.08 points at HoH@3. The gains continue to accumulate across iterations, progressively widening the advantage over Vanilla. Complete task trajectories and recovery cases are provided in the supplementary material.

HoH sustains gains over longer horizons. To test whether gains plateau after three iterations, we run Codex with GPT-5.5 (high) from HoH@3 through HoH@10 on the same 15 FrontierSWE tasks. Figure 6 ranks Vanilla and HoH@1–10 within this run using checkpoint dominance, which differs from the cross-configuration Dominance in Table 1. For each task, the metric is the pairwise win rate over ten comparisons: 1 for a win, 0.5 for a tie, and 0 for a loss. We then macro-average across tasks. The metric rises from 40.33% at HoH@3 to 71.67% at HoH@8, reaches its highest observed value of 73.67% at HoH@9, and is 70.33% at HoH@10—43.00 percentage points above Vanilla (27.33%). Although the trend is not monotone, the later checkpoints retain a large advantage over both Vanilla and HoH@3. In this extended run, we therefore do not observe saturation at the three-iteration horizon, and HoH retains most of its highest observed dominance through HoH@10.

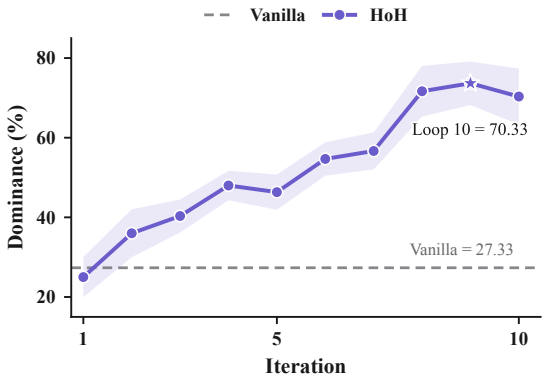

Figure 6 | HoH gains persist at iteration 10. Across 15 FrontierSWE tasks, the band shows ±1 SE; the star marks the best checkpoint and the dashed line Vanilla.

HoH converts additional inference into larger quality gains. Table 2 presents a pass-controlled comparison between HoH and repeated Vanilla continuation using Codex with GPT-5.5 (high). Vanilla Continuation warm-starts from the current artifact and performs additional standard development passes without the HoH protocol. With three development passes, Vanilla Continuation reaches a score of 58.24 using 6.33M tokens per task, whereas HoH reaches 71.52 using 8.41M tokens. Thus, HoH uses 2.08M more tokens but improves the score by an additional 13.28 points. Relative to the one-pass Vanilla baseline, HoH yields 3.77 score points per additional million tokens, compared with 2.32 for Vanilla Continuation. HoH therefore incurs a higher inference cost, but obtains a larger return in final artifact quality. Further details of the pass-controlled experimental design and complete results are provided in the supplementary material.

Method	Dev. Passes	Score	Tokens (M)
Vanilla	1	49.58	2.59
Vanilla Continuation	2	54.99	4.56
Vanilla Continuation	3	58.24	6.33
HoH	1	59.71	2.88
HoH	2	64.84	5.67
HoH	3	**71.52**	8.41

Table 2 | Pass-controlled comparison on GameCraft-Bench. Codex with GPT-5.5 (high); tokens are mean cumulative coding-harness tokens per task.

Ablation Study. To assess the role of HoH's cross-iteration mechanisms, we evaluate three variants on all 45 GameCraft-Bench tasks using Codex with GPT-5.5 (high), with $T = 3$ for each variant. *w/o Plan Update* freezes the first development document for subsequent iterations ($D_t = D_1$ for $t \geq 2$), whereas *w/o Evidence Feedback* replans without the preceding execution evidence. Both retain artifact warm-start. *w/o Warm-Start* retains evidence-conditioned planning but rebuilds the artifact from A_0 in every iteration.

Variant	Score	Tokens (M)
w/o Plan Update	63.39 (−8.13)	7.56
w/o Evidence Feedback	65.23 (−6.28)	7.46
w/o Warm-Start	63.67 (−7.85)	11.12
Full HoH@3	**71.52**	8.41

Table 3 | GameCraft-Bench ablation with Codex and GPT-5.5 (high). Parentheses show score differences from Full HoH@3; tokens are mean cumulative totals per task.

As shown in Table 3, all three variants underperform *Full HoH@3* on every task. Removing plan updates, excluding execution evidence from replanning, and removing warm-start lowers the score by 8.13, 6.28, and 7.85 points, respectively. Without warm-start, token usage also increases from 8.41M to 11.12M per task because of repeated reconstruction. These results show that later HoH iterations benefit from both revising the development document with execution evidence and continuing from the preceding implementation. Detailed ablation protocols and complete per-task results are provided in the supplementary material.

Qualitative Analysis. On GameCraft-Bench, HoH produces more complete and refined game artifacts, with richer gameplay mechanics, clearer visual presentation, and deeper progression. Figure 5 compares gameplay frames from Vanilla and HoH@3 for three tasks from distinct game families. In Momentum Lab, themed terrain and visual cues make the objective and wall-jump route explicit. In Kitchen Rush, distinct pickup, preparation, plating, and disposal stations form a complete and legible restaurant workflow. In Ant Empire, specialist caste counts, seasonal state, and outcome state expose longer-term colony progression. The corresponding *Overall* scores increase from 34.05 to 70.61, from 42.63 to 73.38, and from 65.52 to 87.88, respectively.

5 Case Study: Multi-Day Autonomous Development of a First-Person Shooter

5.1 Case Setup and Development Environment

5.1.1 Task Definition and Run Configuration

5.1.2 Implementation Environment and Tool Layer

5.1.3 Asset Sources and Provenance

5.1.4 Autonomy Boundaries and Human Intervention

5.2 Development Trajectory and Iterative Refinement

5.2.1 Development Phases and Capability Growth

5.2.2 Issue Evolution across Iterations

We track the issue ledger across iterations to summarize how the development loop discovers, resolves, and occasionally reopens implementation problems. Figure 7 provides a provisional visualization of this trajectory.

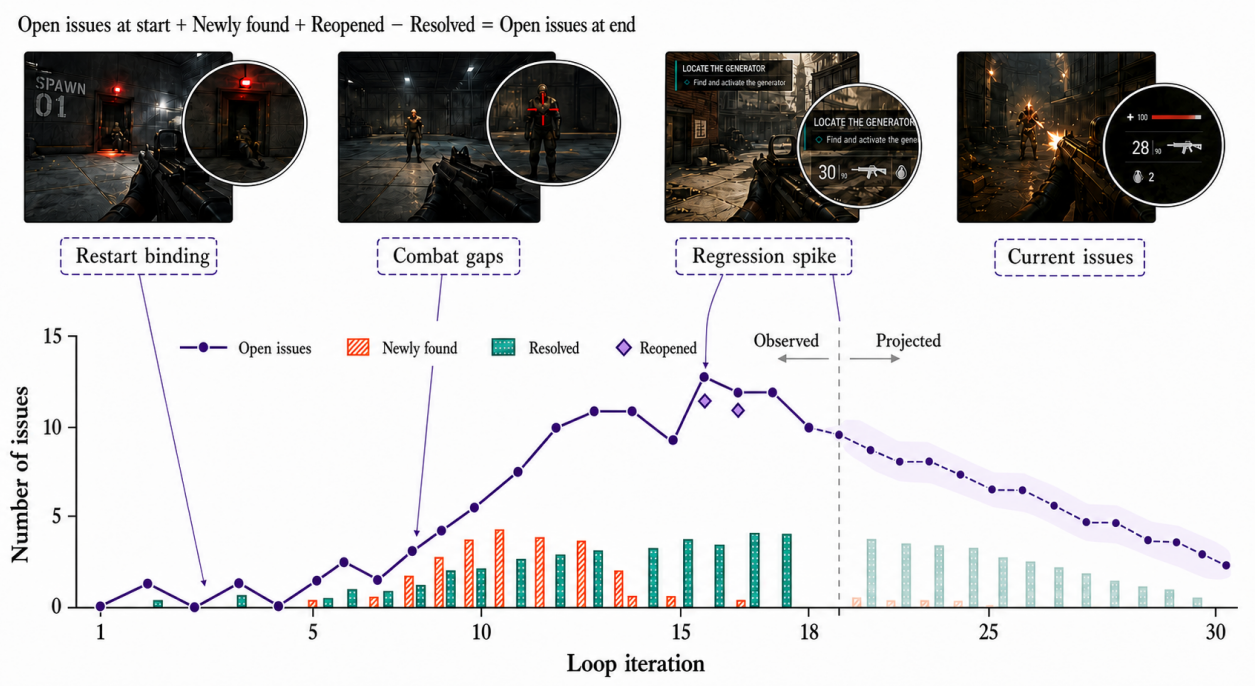

Figure 7 | **Issue evolution during the ongoing Fusepoint case study.** The solid line reports open issues at the start of Loops 1–18; bars show newly found and resolved issues, and diamonds mark reopened issues. The dashed segment and faded bars are projected placeholders through Loop 30. The FPS callouts are synthetic layout placeholders pending replacement with execution screenshots.

5.2.3 Evidence-Driven Refinement and Recovery

5.3 Final Artifact Evaluation and Player Experience

5.3.1 Evaluation Protocol and Runtime Evidence

5.3.2 Requirement Fulfillment and Independent Audit

Table 4 separates three evidence layers for the final artifact: objective deliverability gates, fulfillment of explicitly specified PRD requirements, and independently verified contributions beyond the PRD. This separation prevents optional extensions from masking unmet requirements. Detailed claim-level evidence will be reported in the supplementary material. The displayed statuses are illustrative placeholders and will be replaced after the run and independent audit are complete.

Deliverability gates			Explicit PRD requirements		
Evaluation target	**Reported**	**Verified**	**Evaluation target**	**Reported**	**Verified**
Clean build	✓	✓	World & asset integration	✓	◐ ⚠
Successful launch	✓	✓	Mission flow & objectives	✓	✓
End-to-end completion	✓	◐ ⚠	Combat & enemy AI	✓	◐ ⚠
Failure / restart / replay	✓	✗ ⚠	Controls, UI, & feedback	◐	◐
Repeated-run stability	◐	✗ ⚠	Presentation & polish	◐	◐

Beyond-PRD contributions		
Contribution type	**Reported**	**Verified**
Added functionality	TBD	TBD
Proactive bug fixes	TBD	TBD
Experience improvements	TBD	TBD

✓ Verified complete ◐ Partially complete ✗ Unmet ⚠ Unsupported completion claim

Table 4 | Illustrative placeholder for the independent audit of the Fusepoint artifact. The reported and verified columns compare HoH's self-assessment with the independently audited outcome; warning icons mark unsupported completion claims. Beyond-PRD contributions are counted separately. Detailed evidence and claim-level traces appear in the supplementary material. All entries await the final audit.

5.3.3 Source-Blinded Player Experience Evaluation

We will complement the requirement audit with a small source-blinded playtest involving 3–5 independent evaluators, with both experienced FPS players and participants with computer-science, software-engineering, or game-development backgrounds represented. Evaluators will consent to assessing a research prototype under partial disclosure, receive the same build and task script, complete the main mission once, and then play freely before rating the artifact. Its AI-development provenance will not be disclosed until their ratings are submitted. We will then fully debrief them and allow their anonymized responses to be withdrawn. We use the Player Experience Inventory (PXI) [28]: its 30-item core measures ten constructs—five functional and five psychosocial consequences, with three items per construct—and we retain the official questionnaire's separate three-item Enjoyment outcome. For each evaluator, we average the three items within each outcome; Table 5 reports the across-evaluator mean and sample standard deviation. Given the small sample, we report descriptive statistics and individual ratings rather than significance tests, and do not aggregate the multidimensional instrument into an overall PXI score.

Functional consequences			Psychosocial consequences		
Dimension	**Mean**	**SD**	**Dimension**	**Mean**	**SD**
Ease of Control	TBD	TBD	Meaning	TBD	TBD
Clarity of Goals	TBD	TBD	Curiosity	TBD	TBD
Challenge	TBD	TBD	Mastery	TBD	TBD
Progress Feedback	TBD	TBD	Immersion	TBD	TBD
Audiovisual Appeal	TBD	TBD	Autonomy	TBD	TBD
Enjoyment*	TBD	TBD			

PXI items use a seven-point scale from −3 (strongly disagree) to +3 (strongly agree). Each entry averages its three items per evaluator; SD is the sample standard deviation across evaluators. *Enjoyment is a separate outcome included in the official questionnaire, not a core PXI construct. Detailed per-rater scores and comments appear in the supplementary material.

Table 5 | Placeholder for the source-blinded Fusepoint playtest using PXI. The table covers the ten core PXI constructs and the separate Enjoyment outcome. All scores remain TBD until the independent evaluation is complete; no playtest result is assumed here.

5.4 Discussion and Practical Implications

After the user provided the PRD, planning, implementation, debugging, and verification proceeded without further human input into development decisions. Human intervention was limited to restoring quota or external-service availability. These interventions neither modified the artifact nor redirected the development plan. The autonomy demonstrated here therefore concerns the development process rather than uninterrupted operation of its supporting infrastructure.

Asset production remained a separate bottleneck. Attempts to generate 3D assets were costly and did not reliably reach the visual quality required by the game. The final artifact therefore uses openly licensed assets, including CC0 and CC BY resources. We retain provenance, license metadata, and any required attribution for these assets. In this case, integrating licensed content proved more reliable than generating all production-quality assets end to end.

This longitudinal study provides evidence for end-to-end autonomous game development, but not for comparable development in other software domains. Although the broader benchmarks evaluate repository engineering and program reconstruction, they do not reproduce the same sustained, product-level development process. We will release additional game and software cases on the project website to broaden this evidence base. These future artifacts are not part of the evidence evaluated here.

6 Conclusion

We introduced Harness-of-Harness (HoH), which extends existing coding-agent harnesses to support software development from scratch without modifying their implementations. HoH organizes a fixed harness–model configuration into a continuous planning–coding–testing cycle, carrying evolving artifacts and execution evidence across iterations. Across game development and long-horizon repository-engineering tasks, HoH consistently improves final artifact quality across all three evaluated configurations and continues to benefit from additional iterations. Just as a coding harness structures model operation, HoH structures harness participation in long-horizon development. More broadly, HoH offers a practical path toward end-to-end software development through persistent, evidence-grounded orchestration of coding-agent harnesses.

References

[1] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 73–87. ACM, 2000. doi: 10.1145/336512.336534. https://doi.org/10.1145/336512.336534.

[2] Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei A. Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? In *Advances in Neural Information Processing Systems*, volume 38. Curran Associates, Inc., 2025. https://proceedings.neurips.cc/paper_files/paper/2025/file/b1041e52d3be19f0a9bc491657488e4a-Paper-Datasets_and_Benchmarks_Track.pdf.

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

[4] Evan Chu, Rajan Agarwal, Abishek Thangamuthu, Brendan Graham, Justus Mattern, Freeman Jiang, Paul Cento, Swarnim Jain, Mersad Abbasi, Mohammad Hossein Rezaei, George Wang, Alex Zhang, Simon Guo, Karina Nguyen, Danna Liu, Arash Bidgoli, Aditya Dalmia, Apoorv Dankar, Ashrut Vaddela, Calvin Chen, Keshav Kumar, Kushagra Vaish, Navid Pour, Rishyanth Kondra, Sagar Badiyani, Sidharth Giri, Snagnik Das, Soham Gaikwad, Syed Shah, Vagish Dilawari, and Vishal Agarwal. Frontierswe. *Proximal Blog*, 2026. https://frontierswe.com/blog.

[5] Yue Hu, Yuzhu Cai, Yaxin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and Siheng Chen. Self-evolving multi-agent collaboration networks for software development. In *International Conference on Learning Representations*, pages 23007–23039, 2025. https://proceedings.iclr.cc/paper_files/paper/2025/file/39af4f2f9399122a14ccf95e2d2e7122-Paper-Conference.pdf.

[6] Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. AgentCoder: Multi-agent-based code generation with iterative testing and optimisation, 2023. https://arxiv.org/abs/2312.13010.

[7] 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*, pages 54107–54157, 2024. https://proceedings.iclr.cc/paper_files/paper/2024/file/edac78c3e300629acfe6cbe9ca88fb84-Paper-Conference.pdf.

[8] 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. https://arxiv.org/abs/2512.18470.

[9] Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-end optimization of model harnesses, 2026. https://arxiv.org/abs/2603.28052.

[10] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9): 1060–1076, 1980. doi: 10.1109/PROC.1980.11805. https://doi.org/10.1109/PROC.1980.11805.

[11] Junjie Li, Xi Xiao, Yunbei Zhang, Chen Liu, Lin Zhao, Xiaoying Liao, Yingrui Ji, Janet Wang, Yingqiang Ge, Weijie Xu, Xi Fang, Xiang Xu, Tianchen Zhao, Youngeun Kim, Jihun Hamm, Tianyang Wang, and Chandan Reddy. Agent harness engineering: A survey, 2026. https://openreview.net/pdf?id=eONq7FdiHa.

[12] 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. https://arxiv.org/abs/2604.25850.

[13] Jiacheng Liu, Xiaohan Zhao, Xinyi Shang, and Zhiqiang Shen. Dive into Claude Code: The design space of today's and future AI agent systems, 2026. https://arxiv.org/abs/2604.14228.

[14] Junwei Liu, Chen Xu, Chong Wang, Tong Bai, Weitong Chen, Kaseng Wong, Yiling Lou, and Xin Peng. Towards iterative end-to-end software development: A feature-driven multi-agent framework, 2026. https://arxiv.org/abs/2511.02399. Accepted at the 35th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2026).

[15] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35, 2023. doi: 10.1145/3560815. https://doi.org/10.1145/3560815.

[16] Xinghua Lou, Miguel Lázaro-Gredilla, Antoine Dedieu, Carter Wendelken, Wolfgang Lehrach, and Kevin P. Murphy. AutoHarness: Improving LLM agents by automatically synthesizing a code harness, 2026. https://arxiv.org/abs/2603.03329.

[17] Pengrui Lu, Shiqi Zhang, Yunzhong Hou, Lyumanshan Ye, Chaoyi Huang, Zixi Chen, Ji Zeng, Hantao Jiang, Pengfei Liu, Yiwei Wang, and Ming-Hsuan Yang. ProjDevBench: Benchmarking AI Coding Agents on End-to-End Project Development, 2026. https://arxiv.org/abs/2602.01655.

[18] Tongxu Luo, Rongsheng Wang, Jiaxi Bi, Chenming Xu, Zhengyang Tang, Jianlong Chen, Juhao Liang, Ke Ji, Shuqi Guo, Yuhao Du, Fan Bu, Wenyu Du, Xiaotong Zhang, Kyle Li, Shaobo Wang, Linfeng Zhang, Yuxuan Liu, Xin Lai, Chenxin Li, Yiduo Guo, Zhexin Zhang, Xinyuan Wang, Tianyi Bai, Ziniu Li, and Benyou Wang. GameCraft-Bench: Can Agents Build Playable Games End-to-End in a Real Game Engine?, 2026. https://arxiv.org/abs/2606.17861.

[19] Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

[20] Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. SWT-Bench: Testing and validating real-world bug-fixes with code agents. In Amir Globerson, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub Tomczak, and Cheng Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 81857–81887. Curran Associates, Inc., 2024. doi: 10.52202/079017-2601. https://proceedings.neurips.cc/paper_files/paper/2024/file/94f093b41fc2666376fb1f667fe282f3-Paper-Conference.pdf.

[21] Minh Huynh Nguyen, Thang Phan Chau, Phong X. Nguyen, and Nghi D. Q. Bui. AgileCoder: Dynamic collaborative agents for software development based on agile methodology. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (FORGE)*, pages 156–167. IEEE, 2025. doi: 10.1109/FORGE66646.2025.00026. https://doi.org/10.1109/FORGE66646.2025.00026.

[22] 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. https://arxiv.org/abs/2605.18747.

[23] Gabriel Orlanski, Devjeet Roy, Alexander Yun, Changho Shin, Alex Gu, Albert Ge, Dyah Adila, Nicholas Roberts, Frederic Sala, and Aws Albarghouthi. SlopCodeBench: Benchmarking how coding agents degrade over long-horizon iterative tasks, 2026. https://arxiv.org/abs/2603.24755.

[24] Khouloud Oueslati, Maxime Lamothe, and Foutse Khomh. RefAgent: A Multi-Agent LLM-Based Framework for Automatic Software Refactoring. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering*, 2026. https://arxiv.org/abs/2511.03153.

[25] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems, 2024. https://arxiv.org/abs/2310.08560.

[26] Wenbo Pan, Shujie Liu, Xiangyang Zhou, Shiwei Zhang, Wanlu Shi, Mirror Xu, and Xiaohua Jia. *M**: Every task deserves its own memory harness. *arXiv preprint arXiv:2604.11811*, 2026.

[27] 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, pages 8634–8652, 2023. https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

[28] Vero Vanden Abeele, Katta Spiel, Lennart E. Nacke, Daniel Johnson, and Kathrin Gerling. Development and validation of the player experience inventory: A scale to measure player experiences at the level of functional and psychosocial consequences. *International Journal of Human-Computer Studies*, 135: 102370, 2020. doi: 10.1016/j.ijhcs.2019.102370. https://doi.org/10.1016/j.ijhcs.2019.102370.

[29] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations*, pages 65882–65919, 2025. https://proceedings.iclr.cc/paper_files/paper/2025/file/a4b6ad6b48850c0c331d1259fc66a69c-Paper-Conference.pdf.

[30] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824, 2025. doi: 10.1145/3715754. https://doi.org/10.1145/3715754.

[31] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-Agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pages 50528–50652, 2024. doi: 10.52202/079017-1601. https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

[32] John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. ProgramBench: Can Language Models Rebuild Programs From Scratch?, 2026. https://arxiv.org/abs/2605.03546.

[33] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. https://openreview.net/forum?id=WE_vluYUL-X.

[34] Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-Harness: Harnesses that improve themselves, 2026. https://arxiv.org/abs/2606.09498.

[35] 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 *International Conference on Learning Representations*, 2026. https://openreview.net/forum?id=eC4ygDs02R.

[36] Wenting Zhao, Nan Jiang, Celine Lee, Justin Chiu, Claire Cardie, Matthias Gallé, and Alexander Rush. Commit0: Library generation from scratch. In *International Conference on Learning Representations*, pages 12061–12076, 2025. https://proceedings.iclr.cc/paper_files/paper/2025/file/1fcefa894924bb1688041b7a26fb8aea-Paper-Conference.pdf.

[37] Qixing Zhou, Jiacheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, Dandan Tu, and Zhaoxiang Zhang. FeatureBench: Benchmarking agentic coding for complex feature development. In *International Conference on Learning Representations*, 2026. https://openreview.net/forum?id=41xrZ3uGuI.

[38] 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. https://proceedings.mlr.press/v235/zhuge24a.html.