arXiv:2609.32459v1 [cs.SE] 26 Sep 2026

Beyond the Model: Demystifying Harness Effects in Software Engineering Agents

Haichuan Hu[1] **Quanjun Zhang**[1] **Shengcheng Yu**[2] **Zhifei Chen**[1]
Tianyu Luo[3] **Chunrong Fang**[3] **Zhenyu Chen**[3] **Liang Xiao**[1]

[1]Nanjing University of Science and Technology [2]Technical University of Munich
[3]Nanjing University

huhaichuan2024@gmail.com, quanjunzhang@njust.edu.cn, shengcheng.yu@tum.de
chenzhifei@njust.edu.cn, Ty_L191025@outlook.com, fangchunrong@nju.edu.cn
zychen@nju.edu.cn, xiaoliang@mail.njust.edu.cn

Large Language Model (LLM)-based agents are increasingly used for software engineering tasks, yet their performance is not determined by the base model alone. The agent harness substantially shapes how SE agents interact with repositories, execute actions, and validate solutions. However, the role of harness design remains insufficiently understood, especially across different models, tasks, and harness components. In this paper, we present a systematic empirical study of harness effects in SE agents. We first evaluate two representative harnesses, mini-SWE-agent and OpenCode, with ten models from two prominent open-weight model families, Qwen and DeepSeek, on three benchmarks: SWE-bench Pro, ProgramBench, and GitTaskBench. We then construct NanoHarness, a lightweight modular harness built on top of mini-SWE-agent, and use it to analyze five representative harness components: tool registry, context compression, explicit planning, subagents, and lazy skills. Experimental results show that harness effectiveness depends jointly on model capability and task type. Complex harnesses provide diminishing marginal gains on SWE-style issue repair as model capability improves, but can benefit stronger models on more complex and open-ended repository-level tasks. Component-level analysis on ProgramBench further shows that structured tool use and task-specific subagents provide the most stable improvements, while context compression and general subagents can hurt repository-generation performance. When combined, NanoHarness improves over mini-SWE-agent by 7.37 and 6.21 percentage points on Qwen3.7-Max and DeepSeek-V4-Pro, respectively, recovering most of the gains of product-level harnesses. These findings highlight harness design as a first-class factor in SE-agent performance and provide insights for building more effective and efficient coding agents.

KEYWORDS Large Language Models, Agent Harness, Software Engineering, Code Generation, Coding Agents

DATE September 26, 2026

1 Introduction

Large Language Model (LLM)-based agents are increasingly used to solve Software Engineering (SE) tasks, such as issue resolution [11, 56, 64], code modification [14, 16, 22], and test generation [29, 47]. While much progress has been driven by stronger foundation models (e.g., GPT [1, 42], Claude [4], Gemini [43], DeepSeek [26], Qwen [54]), the performance of coding agents also depends heavily on the model-external infrastructure that surrounds the model, including context retrieval [34, 62], tool use [12, 63], execution environments [51], patch generation [21, 40], and validation [13, 65]. This infrastructure is commonly referred to as the agent harness [30, 68].

Publicly reported results on the SWE-bench Verified leaderboard [20] suggest that harness design can have an impact comparable to model choice. As shown in Figure 1, the gap between the best- and worst-performing harnesses under a fixed model reaches 19.4 percentage points, close to the 22.8-point gap between Claude models under a fixed harness. Epoch AI [9] also points out that harnesses matter as much as model choice, and that a well-designed harness can yield roughly a 20% performance improvement. These motivates a controlled analysis of which harness design choices drive such gains and whether they transfer across models and tasks.

Despite this impact, existing SE research has not yet isolated harness design as an object of systematic study. Recent work [8, 23, 45, 49, 60] has introduced increasingly capable coding agents for specific software engineering scenarios. These agents necessarily make concrete harness-level choices, such as how to retrieve repository context, expose tools, plan edits, run tests, and select final patches. However, these choices are typically evaluated as part of an end-to-end agent system, rather than analyzed as separable design components. As a result, we still lack a clear understanding of which harness components drive improvements, how they interact, and how their effects depend on base models and task types.

To bridge this gap, we conduct an empirical study of harnesses in SE agents. Our study focuses on two complementary perspectives: (1) how models, harnesses, and tasks jointly shape agent performance; and (2) how harness components contribute to performance on general SE tasks.

For the first perspective, we evaluate agents on three rep-

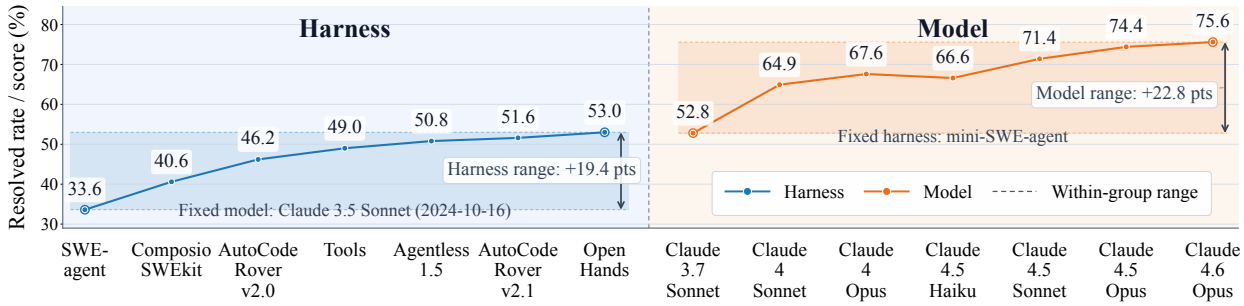


Figure 1: Comparing fixed-model and fixed-harness settings on SWE-bench Verified. Results are collected from the leaderboard.

resentative SE tasks that broadly cover the SE lifecycle: issue repair on SWE-bench Pro [11], repository-level code generation on ProgramBench [57], and repository-centric real-world task solving on GitTaskBench [31]. We study these tasks under two contrasting harness configurations: mini-SWE-agent [56], a lightweight harness with basic command-line interaction capabilities, and OpenCode [2], a product-level harness with a more complex design. We pair these harnesses with two model families, Qwen [54] and DeepSeek [26], selecting five representative models from each family. Overall, this design yields 60 experimental configurations (3 tasks × 2 harnesses × 10 models), enabling a systematic analysis of how models, harnesses, and tasks jointly shape agent performance.

For the second perspective, we follow commercial harness designs (e.g., Claude Code [3] and Codex [33]) to identify five core components of modern harness architectures, and implement each component in a minimal form. Starting from mini-SWE-agent, we incrementally add these plug-and-play components to the existing harness in a building-block manner, creating a controlled harness variant called NanoHarness for component-level analysis. Using ProgramBench as the testbed, we evaluate both the isolated effect of each component and the combined effect of integrating all components. In addition, we collect and analyze both the performance and execution behavior of NanoHarness and its variants, and further conduct trajectory-level analysis.

Experimental results show that harness design substantially shapes SE-agent performance, but its effect is neither uniform nor monotonic. First, harnesses play different roles under different task regimes. On SWE-style issue repair, where workflows are relatively fixed and edits are often localized, complex harnesses mainly compensate for weaker models, and their marginal benefit decreases as model capability improves. In contrast, on newer and more open-ended repository-level tasks such as ProgramBench and GitTaskBench, complex harnesses behave more like capability amplifiers: only sufficiently capable models can reliably exploit the additional workflow complexity.

Second, our component and trajectory analyses show that harness gains come less from adding more scaffolding or spending more tokens, and more from regulating how agents explore and act in repositories. Structured tool use and task-specific subagents provide the most stable improvements, while context compression and generic subagents can hurt repository-generation performance. NanoHarness improves over mini-SWE-agent by 7.37 and 6.21 percentage points on Qwen3.7-Max and DeepSeek-V4-Pro, respectively, and closes most of the gap to product-level harnesses with only controlled prompt growth. These results suggest that effective harnesses improve SE agents by turning raw exploration into structured, task-aligned interaction rather than by increasing model capability alone.

This paper makes the following contributions: (1) **Large-scale Empirical Study.** We conduct a systematic empirical study across three representative benchmarks, using 10 different LLMs from Qwen and DeepSeek, with over **10B** (Qwen) and **20B** (DeepSeek) prompt tokens. (2) **Component-Level Decomposition.** We decompose modern SE-agent harnesses into representative components and build NanoHarness, a lightweight modular harness that enables controlled component-level analysis. (3) **Behavioral Insight.** We reveal how harness mechanisms affect performance and execution behavior, showing that effective harnesses improve SE-agent performance mainly by enabling structured tool use and better-regulated exploration rather than by model improvement alone.

2 Preliminaries

In this section, we briefly introduce the architecture and main components of the modern harness design.

2.1 Architecture

As shown in Figure 2, we present a general architecture of modern agent harnesses. The harness is organized around

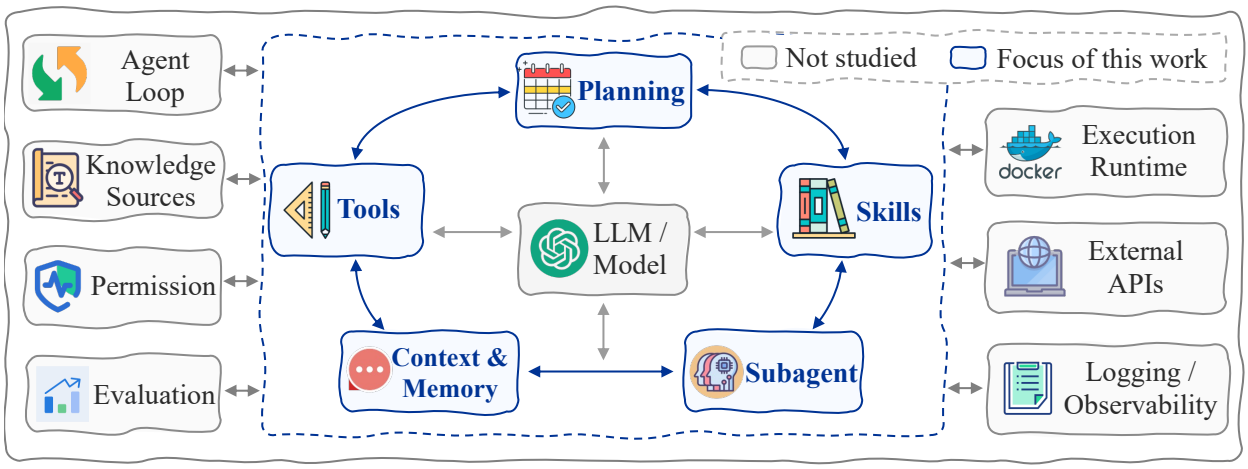


Figure 2: The general architecture of modern agent harnesses.

the model, and we decompose it into 12 modules [30, 68]. The modules highlighted in blue are the focus of this work, as they represent the fundamental capabilities of a harness and have a direct impact on agent performance. The gray modules include supporting components such as environment, permission, logging, observability, knowledge sources, and evaluation. Although these modules are also indispensable parts of a complete agent system, they usually play more engineering-oriented roles or serve as external constraints across different harnesses. Therefore, we do not discuss their internal mechanisms in detail. Next, we introduce the five core modules of the harness.

2.2 Components

2.2.1 Tools

Tools expose external actions such as file inspection, editing, search, shell execution, and testing. Their interface determines what repository state the model can observe and how safely it can act on that state.

2.2.2 Context and Memory

Context and memory determine what task information, observations, code snippets, outputs, and intermediate reasoning remain available during the agent loop. Effective management preserves useful state while limiting irrelevant or redundant history.

2.2.3 Planning

Planning helps the agent decompose a task, track intermediate goals, and decide subsequent actions. Harnesses may implement this through explicit plans, iterative replanning, or lightweight action selection.

2.2.4 Subagents

Subagents delegate parts of a task to specialized roles or execution flows, such as localization, implementation, or verification. This can improve decomposition but also adds coordination and communication overhead.

2.2.5 Skills

Skills encode reusable high-level workflows such as debugging, validation, or recovery from failed attempts. They reduce repetitive decision-making when the procedure is relevant to the current task.

3 Study on SE Agent Harness

3.1 Overall Study Design

Overview. Figure 3 provides an overview of our empirical study on SE agent harnesses. The study is organized into two complementary parts. First, we examine the joint effects of models, harnesses, and tasks by evaluating models with different capability levels under both lightweight and product-level harnesses across SE tasks spanning the software development lifecycle. Second, we conduct a component-level analysis by incrementally adding core harness mechanisms, including tool registry, context compression, explicit planning, subagents, and lazy skills, to a minimal baseline harness. This design allows us to study not only how harnesses interact with model capability and task characteristics, but also how individual components and their combinations affect agent performance, behavior, and efficiency.

Study 1: Measuring Joint Effects of Models, Harnesses, and Tasks. SE research mainly follows three directions: improving coding-oriented models, designing agent harnesses, and building SE tasks and benchmarks.

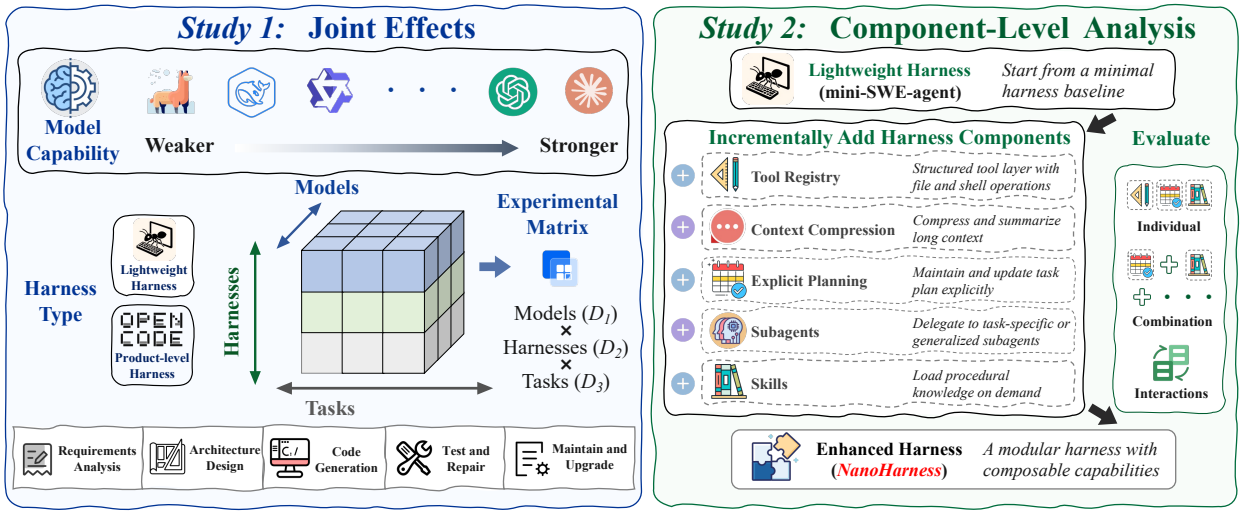


Figure 3: Overview of our agent harness study: (1) joint model–harness–task effects and (2) component-level harness analysis.

Model-oriented studies [39, 55, 59] train models through code pretraining, fine-tuning, reinforcement learning, or inference-time optimization. Harness-oriented studies [7, 48, 52] improve how agents use repositories, tools, tests, and execution environments. Task-oriented and benchmark-oriented studies [38, 46] design realistic SE tasks and evaluation protocols. However, these directions are usually studied with separate focuses, leaving the joint effects of models, harnesses, and tasks underexplored.

To examine these joint effects, we design our first study around the three factors. For models, we select multiple models from the same model family, ranging from weaker to stronger ones, so that performance differences can reflect changes in model scale and capability while reducing variation caused by model lineage. For harnesses, we compare a lightweight basic harness with a more complex product-level harness, which allows us to measure how much the surrounding agent workflow changes SE task performance. For tasks, we choose benchmarks that cover different stages of the software development lifecycle, including requirements analysis [10], architecture design [66], coding [19], testing [29], and validation [13]. By combining these factors in a unified three-dimensional experimental matrix, we evaluate each model under each harness across each task, allowing us to compare their individual effects and analyze how they interact in LLM-based SE agents.

Study 2: Component-Level Harness Analysis. The second study focuses on the internal design of the agent harness. As discussed in Section 2, modern SE agent harnesses can be decomposed into 12 components. To study the mechanisms behind representative components, we implement a modular harness called NanoHarness on top of mini-SWE-agent, where each mechanism can be enabled or disabled independently. Specifically, we instantiate five representative mechanisms: tool registry, context compression, explicit planning, subagents, and lazy skills. This modular design allows us to evaluate their effects on SE tasks both individually and in combination, thereby analyzing how different harness mechanisms contribute to long-horizon performance beyond the choice of model alone.

Tool Registry. We extend the Bash interactive mode of mini-SWE-agent into a tool layer, implemented through a unified registry and native tool-calling mechanism. We introduce four file-system-oriented tools: `read_file`, `write_file`, `edit_file`, and `glob`. `read_file` provides bounded file inspection, `write_file` creates or overwrites files, `edit_file` applies localized textual modifications, and `glob` supports pattern-based file discovery.

Context Compression. We implement a context compression module for long-horizon execution under limited context windows. Oversized tool outputs are persisted to the workspace, with only paths and short previews retained in the prompt. When history grows too long, the module applies message snipping and observation compaction while preserving recent results and high-value outputs. If the context still exceeds the limit, it triggers LLM-based summarization to retain necessary information. Compaction can also be invoked manually or after context-window errors, with logs kept for auditing.

Explicit Planning. The planning module adds a TodoWrite-style lightweight planning mechanism to the agent. When enabled, the agent can call `todo_write` to maintain a short task list, where at most one item may be marked as `in_progress`. Each update replaces the full todo list and is validated by `TodoManager` for item format, status values, list length, and planning consistency. The current plan is rendered back into the prompt and saved

in the trajectory metadata. To avoid stale plans, the agent injects reminders when the plan has not been updated for several rounds.

General/Task-Specific Subagents. The subagent module provides on-demand specialist capabilities while leaving task control, budget management, and final submission to the parent agent. Each subagent runs with fresh history in the shared workspace, uses a restricted tool whitelist via `SubagentRunner`, and returns a structured report through `finish_subagent`. Nested delegation is disabled, and trajectories are logged for auditing and cost analysis.

(1) General SE subagents. The first type consists of software-engineering subagents organized according to the software development lifecycle, including *requirement_analysis*, *architecture_design*, *code_generation*, *test_generation*, and *defect_repair*. Each subagent is assigned an appropriate tool set and returns structured artifacts for the parent agent to integrate.

(2) Task-specific subagents. The second type consists of dedicated profiles for *behavior analysis*, *differential testing*, and *submission review*. These subagents support targeted exploration, validation, and final checking.

Lazy Skills. The skills module reduces prompt overhead by treating procedural knowledge as lazily loaded, callable resources. When enabled, the prompt contains only a compact skill catalog, and the agent can invoke `load_skill(name)` to retrieve detailed guidance when needed. `SkillLoader` builds this catalog from metadata in configured `SKILL.md` files. Loaded skills are returned in a structured wrapper and support implementation, debugging, testing, fuzzing, and final review. The available skills are defined in Table 1.

Table 1: Overview of lazily loaded skills.

Skill	Purpose
Terminal Workflow	Guides terminal-based task solving from exploration to validation and submission.
Minimal Implementation	Encourages small, targeted implementations and rapid validation.
Build Diagnostics	Helps diagnose build, executable, toolchain, and dependency issues.
Test Triage	Analyzes failed tests, logs, exit codes, and interface mismatches.
CLI Fuzzing	Generates bounded edge-case tests and compares CLI behavior.
Submission Review	Checks build reproducibility, executable correctness, and repository cleanliness.

Table 2: Summary of datasets used in our experiments.

Dataset	Task	Language	Size
ProgramBench [57]	Repository generation	Rust, Go, C/C++, Java, Haskell	200
SWE-bench Pro [11]	Issue repair	Python, JS/TS, Go	731
GitTaskBench [31]	Repository task solving	Python	54

3.2 Research Questions

Based on our study objectives, we propose the following three research questions.

RQ1: What is the relationship among models, harnesses, and tasks?

RQ1.1: How does harness performance scale with model capability?
RQ1.2: How do harnesses vary across SE tasks?

RQ2: How do harness components affect SE agents?

RQ2.1: What is the individual effect of each component?
RQ2.2: What is the combined effect of all components?

RQ3: How do harness mechanisms affect agent execution behavior and efficiency?

RQ3.1: How do harness mechanisms change the agent's tool use, context usage, and token usage?
RQ3.2: How do harness mechanisms affect common failure modes in long-horizon SE tasks?

4 Experimental Setup

4.1 Tasks and Datasets

We evaluate harnesses on three complementary SE benchmarks (Table 2): ProgramBench [57], SWE-bench Pro [11], and GitTaskBench [31]. These datasets span key stages of the SE lifecycle, including requirement understanding, architecture design, code generation, testing, and validation. This makes them suitable for assessing harness effectiveness under complex, long-horizon SE tasks. Specifically, ProgramBench targets repository-level code generation, SWE-bench Pro focuses on real-world issue repair, and GitTaskBench evaluates repository-centric task solving.

4.2 Harness Selection

We select mini-SWE-agent [56] and OpenCode [2] as two representative harnesses. mini-SWE-agent serves as the minimal harness setting, offering a concise and basic agent workflow centered on command-line interaction.

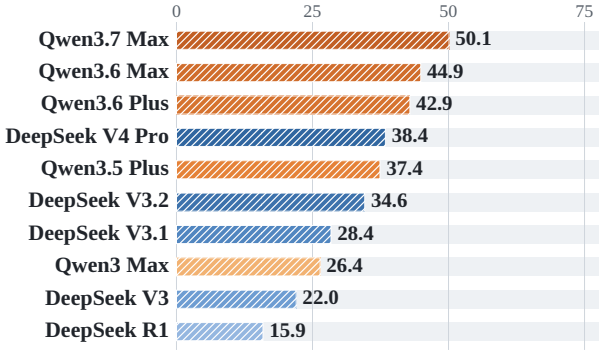


Figure 4: Model coding rankings based on the Coding Index.

It represents a lightweight harness with limited model-external support. OpenCode, in contrast, represents a product-level harness with a more complex design and multiple integrated components for repository interaction, tool use, context handling, and task execution. This contrast enables us to study how harness design complexity influences performance across different models and SE tasks.

4.3 Model Selection

We focus on open-weight model families to support reproducible evaluation and controlled comparisons across models with different capability levels but similar lineage. Therefore, instead of using closed-weight models (e.g., GPT, Claude), we select two widely used open-weight model families, Qwen [6, 54] and DeepSeek [26], and choose five models from each family. We rank the models by strength based on the official scores provided by the Coding Index [5] (Figure 4). This selection and ranking provide an overview of the models' coding capabilities and facilitate comparisons between stronger and weaker models. Here, DeepSeek-R1 refers to the original DeepSeek-R1 release, rather than the later DeepSeek-R1-0528 update, while DeepSeek-V3 refers to DeepSeek-V3-0324. All models are evaluated in non-thinking mode.

Интересно, а почему?

4.4 Implementations

We use a unified execution protocol across all experiments. For model inference, we set the temperature to 0.0 for all models to reduce randomness. For SWE-bench Pro and GitTaskBench, we follow the original evaluation settings and limit each task to 250 agent steps. For ProgramBench, we reduce the step limit from the original 1,000 steps to 300 steps to reduce evaluation cost. Each task is also subject to a 6-hour wall-clock limit and a 3-minute timeout for each command execution. To control evaluation cost in RQ1, we use random subsets for the larger benchmarks: RQ1.1 uses a random sample of 300 instances from SWE-bench Pro, and RQ1.2 uses a random sample of 70 instances from ProgramBench. We use the same sampled instances across all matched model-harness configurations. RQ1.2 uses all 54 GitTaskBench instances. The component-level experiments in RQ2 and RQ3 use the full 200-instance ProgramBench set. SWE-bench Pro and ProgramBench are evaluated in Docker containers, while GitTaskBench is evaluated in a local environment. For containerized tasks, we disable network access with `--network none`, allocate 20 CPUs and 60 GB memory per container, set a 7-hour container lifetime, and disable `SYS_PTRACE`. For GitTaskBench, we isolate the working directory and block operations outside the workspace. For NanoHarness, we set the maximum number of subagent rounds to 50 and limit the planning module to at most 10 todo items. Context compression is enabled with a 350K-token budget, and the most recent 20 messages are preserved when compression is triggered.

Интересно, что при нулевой t наши все равно остается

5 Results and Analysis

5.1 RQ1: Relationship Among Models, Harnesses, and Tasks

5.1.1 RQ1.1-Scaling of Harness Performance with Model Capability

To study how harness-induced gains scale with model capability, we first analyze matched-model results from the SWE-bench Verified [20] leaderboard. As shown in Table 3, more advanced harnesses provide larger improvements over mini-SWE-agent for weaker models, while the margin becomes smaller for stronger models. For example, the gain of OpenHands decreases from +13.60 pp with Claude 3.7 Sonnet to +5.47 pp with Claude 4 Sonnet, and the gain of SWE-agent decreases from +9.60 pp to +4.07 pp under the same model progression. This suggests that harness complexity can substantially improve weaker or mid-level models, but its marginal benefit may be compressed as model capability increases.

Table 3: Matched-model harness improvements over mini-SWE-agent on SWE-bench Verified.

Harness	Model (weak → strong)	mini-SWE	Gain
SWE-agent	Claude 3.7 Sonnet	52.80	+9.60
SWE-agent	Claude 4 Sonnet	64.93	+4.07
OpenHands	Claude 3.7 Sonnet	52.80	+13.60
OpenHands	Claude 4 Sonnet	64.93	+5.47
Tools	Claude 3.7 Sonnet	52.80	+10.40
Tools	Claude 4 Sonnet	64.93	+7.47
Tools	Claude 4 Opus	67.60	+5.60

Our controlled experiments on a random 300-instance sample from SWE-bench Pro [11] further support this

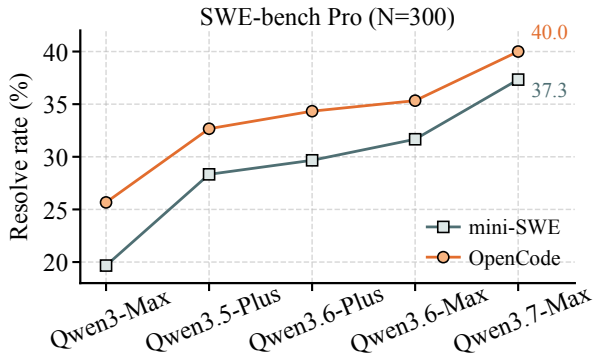


Figure 5: Qwen-model scaling comparison between different harnesses on SWE-bench Pro (N=300).

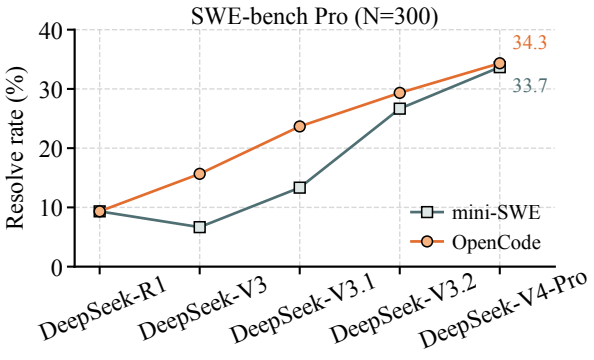


Figure 6: DeepSeek-model scaling comparison between different harnesses on SWE-bench Pro (N=300).

Table 4: Average sample-level statistics of Qwen models.

Harness	Model	Step	Tool	Prompt	Response
mini-SWE	Qwen3-Max	31.12	30.02	369,443	5,110
mini-SWE	Qwen3.5-Plus	89.72	88.05	2,347,080	30,942
mini-SWE	Qwen3.6-Plus	57.81	56.10	1,573,644	66,793
mini-SWE	Qwen3.6-Max	54.15	52.47	1,379,883	27,237
mini-SWE	Qwen3.7-Max	43.58	42.36	770,319	18,732
OpenCode	Qwen3-Max	35.95	39.87	1,069,425	6,442
OpenCode	Qwen3.5-Plus	37.84	57.13	1,702,800	9,505
OpenCode	Qwen3.6-Plus	37.91	61.15	1,499,056	11,132
OpenCode	Qwen3.6-Max	36.49	64.87	1,395,815	11,283
OpenCode	Qwen3.7-Max	37.38	72.23	1,523,661	14,921

trend, but also reveal that the scaling pattern differs across model families. For the Qwen family, the effect is relatively smooth. As model capability increases from Qwen3-Max to Qwen3.7-Max, both mini-SWE-agent and OpenCode improve, while the performance gap between them gradually narrows. Specifically, the OpenCode advantage decreases from about 6 pp for Qwen3-Max to 2.67 pp for Qwen3.7-Max, indicating that stronger Qwen models rely less on the additional scaffolding provided by the product-level harness.

The trajectory statistics in Table 4 provide further evidence for this interpretation. OpenCode keeps the number of agent steps relatively stable across Qwen models, while its number of tool calls increases from 39.87 to 72.23. This suggests that stronger Qwen models can use OpenCode to perform denser tool-assisted exploration within a similar number of decision steps. In contrast, mini-SWE-agent is constrained to roughly one tool call per step, and its step count and token usage first increase and then decrease as model capability improves. This indicates that stronger Qwen models gradually become more selective under the lightweight harness, focusing on fewer but more useful interactions.

DeepSeek exhibits a different scaling pattern, as shown in Figure 6. Unlike Qwen, the harness gain does not decrease smoothly across the whole model sequence.

Интересно, а авторы моделей не фантазируют их не только под свои harness, но и под минимальностью mini-SWE-agent. Ведь это накладывает им ужасные операторы bash + хорошо выглядеть в таких бенчах?

Table 5: Average sample-level statistics of DeepSeek models.

Harness	Model	Step	Tool	Prompt	Response
mini-SWE	DeepSeek-R1	17.71	15.97	127,552	31,280
mini-SWE	DeepSeek-V3	15.90	14.69	104,289	1,886
mini-SWE	DeepSeek-V3.1	58.66	56.66	1,000,415	6,905
mini-SWE	DeepSeek-V3.2	79.99	72.23	1,687,660	13,665
mini-SWE	DeepSeek-V4-Pro	61.77	60.12	1,860,832	29,326
OpenCode	DeepSeek-R1	8.98	7.85	146,731	20,787
OpenCode	DeepSeek-V3	36.42	34.66	1,163,976	5,488
OpenCode	DeepSeek-V3.1	74.91	74.08	2,874,875	10,661
OpenCode	DeepSeek-V3.2	89.42	101.18	2,796,537	19,093
OpenCode	DeepSeek-V4-Pro	31.96	70.42	1,178,990	16,397

OpenCode brings little improvement for DeepSeek-R1, larger gains for intermediate models such as DeepSeek-V3 and DeepSeek-V3.1, and then converges with mini-SWE-agent again for DeepSeek-V4-Pro, where the final scores are 34.3 for OpenCode and 33.7 for mini-SWE-agent. This non-monotonic pattern suggests a threshold effect: models that are too weak may not reliably exploit a complex harness, while sufficiently strong models can solve many instances even with a lightweight harness, reducing the marginal benefit of additional harness mechanisms.

Table 5 further shows that OpenCode changes DeepSeek execution behavior more aggressively than it does for Qwen. For DeepSeek-V3.1 and DeepSeek-V3.2, OpenCode leads to substantially more steps, tool calls, and prompt tokens than mini-SWE-agent, suggesting broader exploration and heavier use of the product-level workflow. However, for DeepSeek-V4-Pro, OpenCode uses fewer steps than mini-SWE-agent while still making more tool calls per step. This indicates that the strongest DeepSeek model can interact with OpenCode in a more compressed and tool-intensive manner, but the resulting performance gain over mini-SWE-agent becomes very small.

Answer to RQ1.1: (1) **The marginal benefit of complex harnesses generally decreases as model capability improves**: on SWE-bench Verified, matched-model gains over mini-SWE-agent drop from +9.60/+13.60 pp

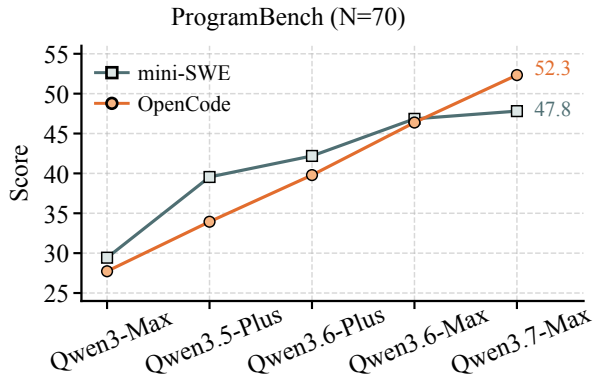


Figure 7: Qwen-model scaling comparison between different harnesses on ProgramBench (N=70).

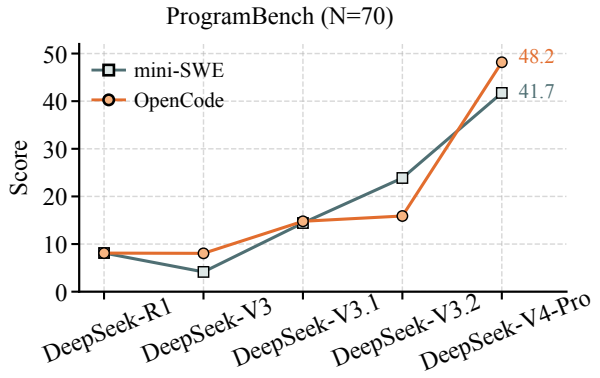


Figure 8: DeepSeek-model scaling comparison between different harnesses on ProgramBench (N=70).

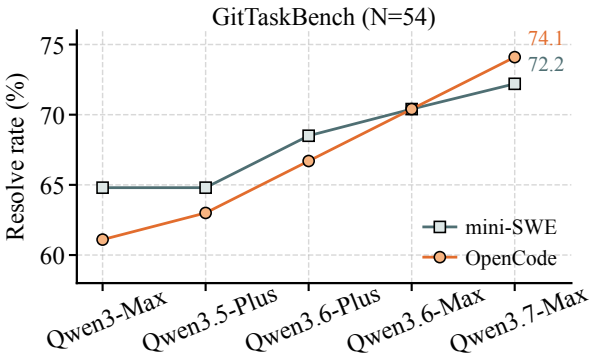


Figure 9: Qwen-model scaling comparison between different harnesses on GitTaskBench (N=54).

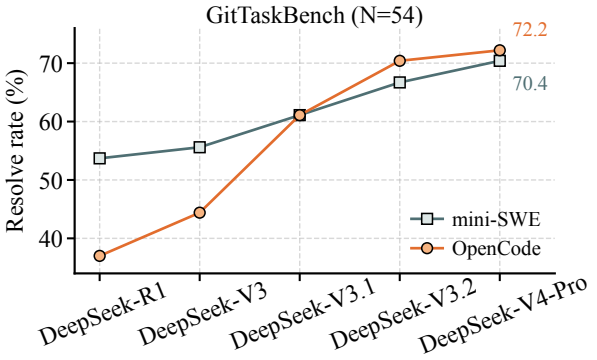


Figure 10: DeepSeek-model scaling comparison between different harnesses on GitTaskBench (N=54).

to +4.07/+5.47 pp, and in Qwen SWE-bench Pro experiments the OpenCode–mini-SWE gap narrows from 6.00 pp to 2.67 pp. (2) **This convergence is not uniform across model families**. Qwen follows a smooth narrowing trend, whereas DeepSeek shows a non-monotonic pattern, with OpenCode helping more on intermediate models (e.g., DeepSeek-V3.1, DeepSeek-V3.2) and nearly converging with mini-SWE on DeepSeek-V4-Pro (34.3% vs. 33.7%).

5.1.2 RQ1.2-Harness Performance on Different SE Tasks.

To verify whether the findings from RQ1.1 are generally applicable, in RQ1.2 we select ProgramBench and GitTaskBench and conduct experiments using the same configuration as RQ1.1. We randomly sample 70 of the 200 instances from ProgramBench, including 20 easy, 21 medium, and 29 hard instances, and use all 54 instances from GitTaskBench.

As shown in Figures 7 and 8, we find that on ProgramBench, OpenCode does not bring greater improvements to weaker models than mini-SWE-agent. Only the latest models (e.g., Qwen3.7-Max and DeepSeek-V4-Pro) can consistently leverage the complex harness to complete tasks more effectively. On GitTaskBench (Figures 9 and 10), we also observe a similar phenomenon: Qwen3.6-Max and DeepSeek-V3.1 are the respective turning points within their model families at which models begin to leverage a complex harness such as OpenCode.

The above results indicate that model capability is not the only factor affecting the effectiveness of a harness; task type and difficulty are also important factors. Compared with ProgramBench and GitTaskBench, SWE-bench Pro has a relatively fixed workflow, and code modifications typically involve only functions in a small number of files. In contrast, ProgramBench is a repository-generation task from scratch, often requiring the model to generate tens of thousands of lines of code while covering fine-grained functionality. The difficulty of GitTaskBench lies in its task flexibility: it involves multiple objectives and requires the model to independently understand the task goal, leverage repository information, and design an execution workflow. In addition, given the growing industry attention to SWE-style tasks in recent years, models may have received task-specific training, which could make them inherently stronger on this type of task and thereby reduce their dependence on the harness.

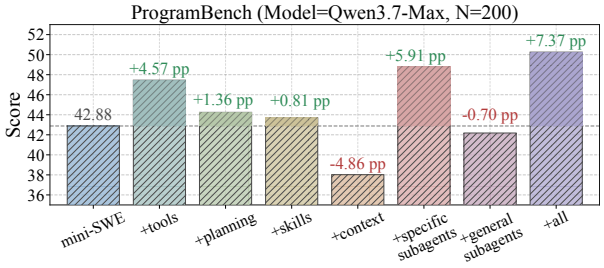


Figure 11: Effect of different harness components on ProgramBench scores using Qwen3.7-Max.

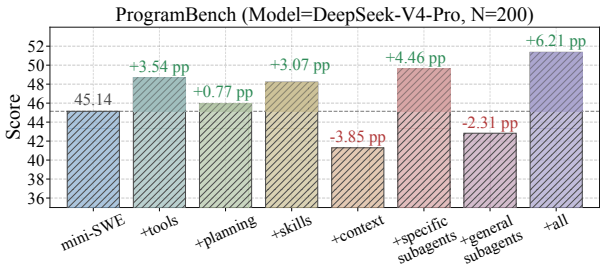


Figure 12: Effect of different harness components on ProgramBench scores using DeepSeek-V4-Pro.

Answer to RQ1.2: The effectiveness of a harness is influenced not only by model capability, but also by task complexity and task type. For SWE-bench Pro, where the workflow is relatively fixed, the amount of code modification is small, and the task has received sustained attention over time, models generally demonstrate strong adaptability. As model capability improves, the additional gains brought by a complex harness (e.g., OpenCode) gradually diminish. In contrast, for newer and more complex SE tasks such as ProgramBench and GitTaskBench, complex harnesses can provide greater benefits to stronger models (e.g., Qwen3.7-Max and DeepSeek-V4-Pro).

5.2 RQ2: Component-Level Effect Analysis

In RQ1, we treat the harness as a whole and analyze its performance across different SE tasks. Although the results show that complex harnesses (e.g., OpenCode) can improve performance compared with basic harnesses (e.g., mini-SWE-agent), the sources of these improvements remain unclear. Therefore, in RQ2, we decompose and combine representative components of the harness to construct the lightweight and modular NanoHarness. We then conduct ablation experiments on the full ProgramBench set with NanoHarness, using two advanced LLMs, Qwen3.7-Max and DeepSeek-V4-Pro.

5.2.1 RQ2.1-Individual Effect

Figures 11 and 12 show that different harness components contribute unevenly to agent performance. Among the individual components, the tool registry and task-specific subagents bring the most consistent improvements across both models. For Qwen3.7-Max, adding the tool registry improves the ProgramBench score by 4.57 percentage points, while task-specific subagents bring the largest single-component gain of 5.91 percentage points. For DeepSeek-V4-Pro, the same two components also provide strong gains of 3.54 and 4.46 points, respectively. These results suggest that structured repository operations and task-oriented delegation are particularly useful for repository-generation tasks, where agents must inspect files, create or modify multiple modules, and repeatedly validate implementation behavior.

Planning and lazy skills also improve performance, but their effects are more moderate and model-dependent. Explicit planning yields small gains for both Qwen3.7-Max and DeepSeek-V4-Pro, indicating that lightweight task lists can help organize long-horizon execution, but planning alone is insufficient to substantially improve repository generation. Lazy skills show a larger benefit for DeepSeek-V4-Pro than for Qwen3.7-Max, suggesting that procedural guidance is more helpful when the model can effectively decide when and how to invoke it.

In contrast, context compression and general subagents degrade performance for both models. Context compression reduces the score by 4.86 percentage points for Qwen3.7-Max and 3.85 percentage points for DeepSeek-V4-Pro, because ProgramBench requires preserving detailed requirements, interface constraints, and implementation states over long trajectories. Aggressive compression may remove information that is still useful for later coding and debugging. General subagents also underperform, indicating that generic SE delegation may introduce coordination overhead or produce artifacts that are not directly aligned with the fine-grained requirements of repository generation.

Answer to RQ2.1: Individual harness components have heterogeneous effects. Tool registry and task-specific subagents provide the most stable and substantial gains, planning and lazy skills offer smaller or model-dependent improvements, while context compression and general subagents can hurt performance on repository-generation tasks.

5.2.2 RQ2.2-Combined Effect

When all components are combined, NanoHarness achieves the best performance among the variants for both models. For Qwen3.7-Max, the full configuration

improves the score from 42.88 to 50.25, yielding a 7.37-point gain over mini-SWE-agent. For DeepSeek-V4-Pro, it improves the score from 45.14 to 51.35, corresponding to a 6.21-point gain. These improvements are larger than those of any single component, showing that multiple harness mechanisms can complement one another when integrated into a unified workflow.

However, the combined improvement is not simply the sum of all individual gains. Some components, such as context compression and general subagents, have negative effects in isolation, and their interaction with stronger components may introduce additional overhead. This indicates that harness components are not independent: their benefits depend on how they are coordinated within the agent loop. Nevertheless, the full NanoHarness configuration still consistently improves both models, suggesting that positive components such as structured tools, task-specific delegation, and procedural skills can dominate the overall effect when combined properly.

Compared with product-level harnesses, NanoHarness also achieves competitive performance while remaining lightweight and modular. On Qwen3.7-Max, NanoHarness reaches 50.25, reducing the gap to OpenCode (51.68) and Claude Code (52.33) to 1.43 and 2.08 percentage points, respectively. On DeepSeek-V4-Pro, NanoHarness achieves 51.35, which is also close to OpenCode (52.48) and Claude Code (52.76), with gaps of only 1.13 and 1.41 percentage points. These results show that although product-level harnesses still obtain the best scores, a lightweight modular harness composed of representative components can recover most of their performance gains. This suggests that the benefit of complex harnesses does not necessarily come from their full engineering stack; instead, a small set of carefully selected and coordinated components can account for a substantial portion of the improvement.

Answer to RQ2.2: The full NanoHarness configuration achieves the strongest performance among variants, improving over mini-SWE-agent on both models and outperforming any single component. Although NanoHarness still slightly trails product-level harnesses such as OpenCode and Claude Code, it closes most of the gap with a lightweight modular design, suggesting that a compact set of representative components can reproduce much of the benefit of complex harnesses.

5.3 RQ3: Effects of Harness Mechanisms on Agent Behavior and Efficiency

In RQ3, we collect statistics on the performance and cost of each configuration in RQ2, and further analyze execution trajectories to examine how harness components affect agent behavior. Specifically, RQ3.1 analyzes the differences in agent steps, tool calls, and token usage across different configurations of NanoHarness, while RQ3.2 measures the distribution patterns of probing behavior and uses case studies to illustrate shifts in failure modes.

Table 6: Comparing NanoHarness with other product-level harnesses on ProgramBench.

Harness	Qwen3.7-Max	DeepSeek-V4-Pro
mini-SWE-agent	42.88	45.14
NanoHarness	50.25 (+7.37%)	51.35 (+6.21%)
OpenCode	51.68 (+8.80%)	52.48 (+7.34%)
ClaudeCode	52.33 (+9.45%)	52.76 (+7.62%)

5.3.1 RQ3.1-Effects on Tool Use, Context Usage, and Token Usage

Table 7 shows three main behavior patterns. First, tool-oriented mechanisms increase repository interaction: the tool registry modestly raises tool calls, and task-specific subagents increase tool calls by 115.76% for Qwen3.7-Max and 73.10% for DeepSeek-V4-Pro. Since task-specific subagents improve performance while general subagents do not, additional tool use appears beneficial only when it is directed toward the task. Second, context compression reduces cost but hurts performance. It cuts prompt tokens by 54.29% and 77.02% for the two models, yet RQ2 shows corresponding score drops of 4.86 and 3.85 percentage points, suggesting that compressed histories may lose requirements or debugging state needed later. Third, the full NanoHarness configuration improves performance with controlled prompt growth: although tool calls increase by 131.18% and 83.58%, prompt-token growth is much smaller (+16.37% and +7.26%), while scores improve by 7.37 and 6.21 percentage points over mini-SWE-agent. This suggests that the combined harness gains come from more structured interaction rather than simply spending more prompt budget.

Answer to RQ3.1: Harness mechanisms affect execution behavior in three main ways. Tools and subagents increase tool use, context compression reduces cost but may hurt performance, and the full NanoHarness configuration improves performance through more structured interaction with controlled prompt growth.

5.3.2 RQ3.2-Effects on Failure Modes

Through manual inspection of failed and low-scoring trajectories produced by mini-SWE-agent, we categorize failures by probing count and validation behavior and

Table 7: Average sample-level statistics of Qwen3.7-Max and DeepSeek-V4-Pro with different harness components on ProgramBench. Percentages in parentheses indicate relative changes compared with mini-SWE of the same model.

Model	Configuration	Step	Tool	Prompt	Response
Qwen3.7-Max	mini-SWE	104.13	104.14	3,978,148	46,763
Qwen3.7-Max	+ tools	105.89 (+1.69%)	120.87 (+16.06%)	4,299,572 (+8.08%)	44,366 (-5.13%)
Qwen3.7-Max	+ planning	102.19 (-1.86%)	104.35 (+0.20%)	4,817,378 (+21.10%)	65,679 (+40.45%)
Qwen3.7-Max	+ skills	108.69 (+4.38%)	109.20 (+4.86%)	4,503,568 (+13.21%)	47,913 (+2.46%)
Qwen3.7-Max	+ context	61.77 (-40.68%)	62.06 (-40.41%)	1,818,344 (-54.29%)	26,577 (-43.17%)
Qwen3.7-Max	+ specific subagents	141.29 (+35.69%)	224.69 (+115.76%)	4,270,462 (+7.35%)	72,240 (+54.48%)
Qwen3.7-Max	+ general subagents	99.06 (-4.87%)	157.40 (+51.14%)	4,372,546 (+9.91%)	67,919 (+45.24%)
Qwen3.7-Max	+ all (NanoHarness)	133.68 (+28.38%)	240.75 (+131.18%)	4,629,172 (+16.37%)	64,751 (+38.47%)
DeepSeek-V4-Pro	mini-SWE	142.19	181.84	10,065,204	91,370
DeepSeek-V4-Pro	+ tools	164.11 (+15.41%)	204.20 (+12.29%)	11,979,965 (+19.02%)	86,982 (-4.80%)
DeepSeek-V4-Pro	+ planning	140.66 (-1.07%)	184.27 (+1.33%)	10,418,872 (+3.51%)	91,832 (+0.51%)
DeepSeek-V4-Pro	+ skills	147.85 (+3.98%)	179.22 (-1.44%)	10,415,011 (+3.48%)	90,942 (-0.47%)
DeepSeek-V4-Pro	+ context	54.17 (-61.91%)	56.61 (-68.87%)	2,312,701 (-77.02%)	33,747 (-63.07%)
DeepSeek-V4-Pro	+ specific subagents	190.45 (+33.94%)	314.77 (+73.10%)	9,104,564 (-9.54%)	118,453 (+29.64%)
DeepSeek-V4-Pro	+ general subagents	169.54 (+19.24%)	332.14 (+82.65%)	15,284,596 (+51.86%)	145,124 (+58.83%)
DeepSeek-V4-Pro	+ all (NanoHarness)	203.53 (+43.14%)	333.84 (+83.58%)	10,795,507 (+7.26%)	103,443 (+13.21%)

find that models mainly exhibit two failure modes on ProgramBench. (1) **Excessive probing.** At the early exploration stage, the agent blindly performs a large number of fine-grained behavioral probes on the reference binary, but fails to cover the core functionality of the target program. As a result, it exhausts the budget and finishes the task hastily. (2) **Insufficient probing.** The agent assumes that it has already understood the core functionality of the program after only a small number of probes (e.g., ≤ 20), and after implementing the program, it validates the solution using only a few smoke tests. As shown in Figure 13, we compare the distribution of probing times for Qwen3.7-Max under different harness configurations. We observe that the components with the largest performance gains, i.e., task-specific subagents, substantially reduce the tendency of mini-SWE-agent to perform either excessive or insufficient probing. Building on these improvements, NanoHarness further combines the strengths of different components. It not only reduces extreme probing behaviors, but also learns to use more probes to explore the target program behavior more comprehensively, leading to a stable rightward shift in the overall distribution of probing counts.

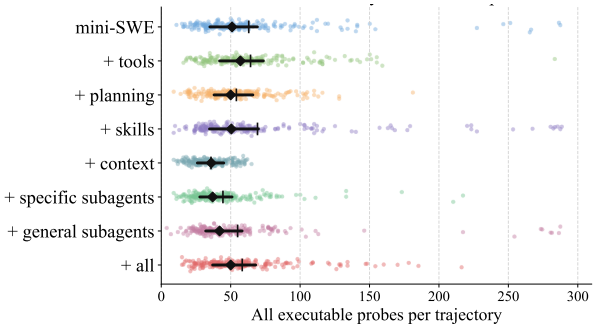


Figure 13: Distribution of probing times (Qwen3.7-Max).

Two cases illustrate how NanoHarness mitigates these extremes. For excessive probing, `jqlang__jq.b33a763` shows how NanoHarness reduces unproductive exploration: mini-SWE-agent issued 133 reference-binary probes and passed only 1602/6491 tests (24.68%), whereas NanoHarness reduced the total number of reference probes to 43 and improved the result to 4425/6491 tests (68.17%). For insufficient probing, `lfos__calcurse.49180d5` shows the opposite correction: mini-SWE-agent made only 3 reference probes before implementation and passed 1231/1994 tests (61.74%), while NanoHarness expanded exploration to 45 reference probes and improved the score to 1499/1994 tests (75.18%). These examples suggest that NanoHarness does not simply increase or decrease probing uniformly; instead, it regularizes exploration by reducing wasteful probing when the agent is stuck and increasing behavioral coverage when the agent is prematurely confident.

Answer to RQ3.2: Harness mechanisms affect failure modes by regularizing probing behavior. On ProgramBench, effective components such as task-specific subagents reduce both excessive low-value probes and insufficient shallow validation. By combining these components, NanoHarness enables more stable behavioral exploration and helps avoid wasteful probing and premature implementation.

6 Related Work

6.1 LLM-based Agents for Software Engineering

LLM-based agents have been widely studied for code generation [18, 27, 69], repository-level problem solving [20, 61], and multi-role software development workflows such as ChatDev [36] and MetaGPT [15]. Other work improves agentic coding through planning, retrieval, debugging, and tool feedback [17, 24, 28]. For repair and issue resolution, SWE-bench [20] has motivated systems such as SWE-agent [56], AutoCodeRover [67], RepairAgent [8], and Agentless [53], while newer benchmarks extend evaluation to broader repository-level tasks [31, 57].

These studies demonstrate the value of agentic workflows, but they usually evaluate complete systems, benchmarks, or task-specific pipelines. As a result, the effects of the model, harness, and task are often entangled. Our work complements them by treating harness design as the main research object and measuring how harnesses and their components affect SE-agent performance across models and tasks.

6.2 Agent Harness

Recent studies emphasize that agent performance depends not only on the base model, but also on the surrounding harness or external scaffolding [25, 30, 32, 35, 68]. General agent frameworks such as ReAct [58], ToolLLM [37], Reflexion [41], Voyager [44], and AutoGen [50] show that tool use, memory, planning, reflection, skills, and multi-agent coordination can improve long-horizon problem solving.

In SE, harnesses must support repository navigation, editing, command execution, testing, and recovery. Existing systems introduce mechanisms such as agent-computer interfaces [56], repository search and fault localization [8, 67], and localization-repair-validation pipelines [53]. However, these mechanisms are usually evaluated as parts of complete agents. Our work isolates representative harness components and analyzes their performance, behavioral, and efficiency effects under controlled model and task settings.

7 Threats to Validity

Internal Threat. The main internal threat stems from our limited selection of harness components: we study five representative components, which may omit other relevant mechanisms. We plan to extend the evaluation to more components in future work. A second threat arises from the coverage of model families, as we use only Qwen and DeepSeek. However, this is mitigated by covering 10 most representative variants, with total prompt-side token usage exceeding **10B** (Qwen) and **20B** (DeepSeek) at an estimated cost of over **$7,000** USD.

External Threat. The main external threat stems from our dataset selection. Our primary experiments are conducted on ProgramBench, so the conclusions may not directly generalize to other datasets. Nevertheless, ProgramBench represents a recent and challenging benchmark for repository-level software generation, covering the complete development lifecycle from requirement understanding to artifact generation. Therefore, we believe ProgramBench provides a meaningful and comprehensive testbed for evaluating the studied harnesses.

8 Conclusion

This paper studies harness design as a first-class factor in LLM-based SE agents. Across three benchmarks, we show that harness gains depend on model capability and task type. Through NanoHarness, we further identify which components matter: structured tools and task-specific subagents improve performance, while context compression and general subagents can hurt. Our trajectory analysis shows that effective harnesses regularize exploration and tool use, suggesting that future SE-agent design should jointly consider models, harnesses, tasks, and execution behavior.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] Anomaly. Opencode: The open source ai coding agent. https://github.com/anomalyco/opencode, 2026. Accessed: 2026-06-11.

[3] Anthropic. Claude Code. https://docs.anthropic.com/en/docs/claude-code/overview. Accessed: 2026-06-23.

[4] Anthropic. The claude 3 model family: Opus, sonnet, haiku. https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf, March 2024. Model card.

[5] Artificial Analysis. Artificial Analysis Coding Index. https://artificialanalysis.ai/models/capabilities/coding, 2026. Accessed: 2026-06-30.

[6] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

[7] Islem Bouzenia and Michael Pradel. You name it, i run it: An llm agent to execute tests of arbitrary projects. *Proceedings of the ACM on Software Engineering*, 2:1054 – 1076, 2024. URL https://api.semanticscholar.org/CorpusID:274763325.

[8] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. Repairagent: An autonomous, llm-based agent for program repair. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 2188–2200. IEEE, 2025.

[9] Florian Brand and Jean-Stanislas Denain. What skills does swe-bench verified evaluate?, 2025. URL https://epoch.ai/publications/what-skills-does-swe-bench-verified-evaluate. Accessed: 2026-06-30.

[10] Senay Tuna Demirel and Resul Das. Software requirement analysis: Research challenges and technical approaches. In *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*, pages 1–6. IEEE, 2018.

[11] Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

[12] Hanxing Ding, Shuchang Tao, Liang Pang, Zihao Wei, Jinyang Gao, Bolin Ding, Huawei Shen, and Xueqi Cheng. Toolcoder: A systematic code-empowered tool learning framework for large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 17876–17891, 2025.

[13] Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. Codescore: Evaluating code generation by learning code execution. *ACM Transactions on Software Engineering and Methodology*, 34(3):1–22, 2025.

[14] Lishui Fan, Jiakun Liu, Zhongxin Liu, David Lo, Xin Xia, and Shanping Li. Exploring the capabilities of llms for code-change-related tasks. *ACM Transactions on Software Engineering and Methodology*, 34(6):1–36, 2025.

[15] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Steven Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, volume 2024, pages 23247–23275, 2024.

[16] Kai Huang, Jian Zhang, Xinlei Bao, Xu Wang, and Yang Liu. Comprehensive fine-tuning large language models of code for automated program repair. *IEEE Transactions on Software Engineering*, 51(4):904–928, 2025.

[17] Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4912–4944, 2024.

[18] Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, volume 2025, pages 58791–58831, 2025.

[19] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–72, 2026.

[20] 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*, volume 2024, pages 54107–54157, 2024.

[21] Youngjoon Kim, Sunguk Shin, Hyoungshick Kim, and Jiwon Yoon. Logs in, patches out: Automated vulnerability repair via {Tree-of-Thought}{LLM} analysis. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 4401–4419, 2025.

[22] Fengjie Li, Jiajun Jiang, Jiajun Sun, and Hongyu Zhang. Hybrid automated program repair by combining large language models and program analysis. *ACM Transactions on Software Engineering and Methodology*, 34(7):1–28, 2025.

[23] Han Li, Yuling Shi, Shaoxin Lin, Xiaodong Gu, Heng Lian, Xin Wang, Yantao Jia, Tao Huang, and Qianxiang Wang. Swe-debate: Competitive multi-agent debate for software issue resolution. *arXiv preprint arXiv:2507.23348*, 2025.

[24] Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, et al. Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 3603–3614, 2024.

[25] 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.

[26] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[27] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in neural information processing systems*, 36:21558–21572, 2023.

[28] Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *International Conference on Learning Representations*, volume 2024, pages 47832–47850, 2024.

[29] Andrea Lops, Fedelucio Narducci, Azzurra Ragone, Michelantonio Trizio, and Claudio Bartolini. A system for automated unit test generation using large language models and assessment of generated test suites. In *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 29–36. IEEE, 2025.

[30] Qianyu Meng, Yanan Wang, Liyi Chen, Qimeng Wang, Chengqiang Lu, Wei Wu, Yan Gao, Yi Wu, and Yao Hu. Agent harness for large language model agents: A survey. 2026.

[31] Ziyi Ni, Huacan Wang, Shuo Zhang, Shuo Lu, Ziyang He, Zhenheng Tang, Sen Hu, Bo Li, Chen Hu, Binxing Jiao, et al. Gittaskbench: A benchmark for code agents solving real-world tasks through code repository leveraging. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 32564–32572, 2026.

[32] Xuying Ning, Katherine Tieu, Dongqi Fu, Tianxin Wei, Zihao Li, Yuanchen Bei, Jiaru Zou, Mengting Ai, Zhining Liu, Ting-Wei Li, et al. Code as agent harness. *arXiv preprint arXiv:2605.18747*, 2026.

[33] OpenAI. Codex: AI Coding Partner from OpenAI. https://openai.com/codex/. Accessed: 2026-06-23.

[34] Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. Repograph: Enhancing ai software engineering with repository-level code graph. In *International Conference on Learning Representations*, volume 2025, pages 30098–30121, 2025.

[35] Linyue Pan, Lexiao Zou, Shuo Guo, Jingchen Ni, and Hai-Tao Zheng. Natural-language agent harnesses. *arXiv preprint arXiv:2603.25723*, 2026.

[36] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pages 15174–15186, 2024.

[37] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *International Conference on Learning Representations*, volume 2024, pages 9695–9717, 2024.

[38] Wasique Islam Shafin, Md Nakhla Rafi, Zhenhao Li, and Tse-Hsun Chen. Evaluating software process models for multi-agent class-level code generation. *ArXiv*, abs/2511.09794, 2025. URL https://api.semanticscholar.org/CorpusID:282992209.

[39] Ye Shang, Quanjun Zhang, Chunrong Fang, Siqi Gu, Jianyi Zhou, and Zhenyu Chen. A large-scale empirical study on fine-tuning large language models for unit testing. *Proceedings of the ACM on Software Engineering*, 2:1678 – 1700, 2024. URL https://api.semanticscholar.org/CorpusID:274982569.

[40] Miaomiao Shao, Yuxin Ding, Cuiyun Gao, Junru Wang, and Guoqing Zhu. Fix pattern-aware vulnerability patch generation via in-context learning. *ACM Transactions on Software Engineering and Methodology*, 2026.

[41] 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.

[42] Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

[43] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[44] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

[45] Haoran Wang, Zhenyu Hou, Yao Wei, Jie Tang, and Yuxiao Dong. Swe-dev: Building software engineering agents with training and inference scaling. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 3742–3761, 2025.

[46] Jian Wang, Xiaofei Xie, Qiang Hu, Shangqing Liu, Jiongchi Yu, Jiaolong Klong, and Yi Li. Defects4c: Benchmarking large language model repair capability with c/c++ bugs. *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 254–265, 2025. URL https://api.semanticscholar.org/CorpusID:282058062.

[47] Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. Testeval: Benchmarking large language models for test case generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3547–3562, 2025.

[48] Xinchen Wang, Pengfei Gao, Xiangxin Meng, Chao Peng, Ruida Hu, Yun Lin, and Cuiyun Gao. Aegis: An agent-based framework for bug reproduction from issue descriptions. *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, 2025. URL https://api.semanticscholar.org/CorpusID:279329351.

[49] Xinchen Wang, Pengfei Gao, Xiangxin Meng, Chao Peng, Ruida Hu, Yun Lin, and Cuiyun Gao. Aegis: An agent-based framework for bug reproduction from issue descriptions. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pages 331–342, 2025.

[50] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First conference on language modeling*, 2024.

[51] Zhiheng Xi, Yiwen Ding, Wenxiang Chen, Boyang Hong, Honglin Guo, Junzhe Wang, Xin Guo, Dingwen Yang, Chenyang Liao, Wei He, et al. Agentgym: Evaluating and training large language model-based agents across diverse environments. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 27914–27961, 2025.

[52] Chun Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *ArXiv*, abs/2407.01489, 2024. URL https://api.semanticscholar.org/CorpusID:270870279.

[53] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

[54] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[55] Boyang Yang, Haoye Tian, Jiadong Ren, Hongyu Zhang, Jacques Klein, Tégawendé F. Bissyandé, Claire Le Goues, and Shunfu Jin. Morepair: Teaching llms to repair code via multi-objective fine-tuning. *ACM Transactions on Software Engineering and Methodology*, 35:1 – 38, 2024. URL https://api.semanticscholar.org/CorpusID:269282730.

[56] John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

[57] John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, et al. Programbench: Can language models rebuild programs from scratch? *arXiv preprint arXiv:2605.03546*, 2026.

[58] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

[59] Lei Yu, Zhirong Huang, Hang Yuan, Shiqi Cheng, Li Yang, Fengjun Zhang, Chenjie Shen, Jiajia Ma, Jingyuan Zhang, Junyi Lu, and Chun Zuo. Smart-llama-dpo: Reinforced large language model for explainable smart contract vulnerability detection. *Proceedings of the ACM on Software Engineering*, 2:182 – 205, 2025. URL https://api.semanticscholar.org/CorpusID:279474218.

[60] Zheng Yu, Ziyi Guo, Yuhang Wu, Jiahao Yu, Meng Xu, Dongliang Mu, Yan Chen, and Xinyu Xing. {PATCHAGENT}: A practical program repair agent mimicking human expertise. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 4381–4400, 2025.

[61] Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Li Aoyan, Lu Chen, Xiaojian Zhong, et al. Multi-swe-bench: A multilingual benchmark for issue resolving. *Advances in Neural Information Processing Systems*, 38, 2026.

[62] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, 2023.

[63] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13643–13658, 2024.

[64] Quanjun Zhang, Chen Gao, Yule Han, Ye Shang, Chunrong Fang, Zhenyu Chen, and Liang Xiao. Sgagent: Suggestion-guided llm-based multi-agent framework for repository-level software repair. *ArXiv*, abs/2602.23647, 2026. URL https://api.semanticscholar.org/CorpusID:286171479.

[65] Quanjun Zhang, Ye Shang, Haichuan Hu, Chunrong Fang, Zhenyu Chen, and Liang Xiao. Compass: Contrastive learning for automated patch correctness assessment in program repair. *Empirical Software Engineering*, 31(6):157, 2026.

[66] Yiran Zhang, Ruiyin Li, Peng Liang, Weisong Sun, and Yang Liu. Knowledge-based multi-agent framework for automated software architecture design. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pages 530–534, 2025.

[67] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1592–1604, 2024.

[68] Chenyu Zhou, Huacan Chai, Wenteng Chen, Zihan Guo, Rong Shan, Yuanyi Song, Tianyi Xu, Yingxuan Yang, Aofan Yu, Weiming Zhang, et al. Externalization in llm agents: A unified review of memory, skills, protocols and harness engineering. *arXiv preprint arXiv:2604.08224*, 2026.

[69] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations*, volume 2025, pages 66602–66656, 2025.