ALPHADIANA: A MEASUREMENT SYSTEM FOR AGENTS IN VERIFIABLE REASONING TASKS

Anonymous authors
Paper under double-blind review

ABSTRACT

Agents pair reasoning models with harnesses that organize model calls, execute tools, load skill instructions, and retain history in memory. Understanding agent performance requires linking execution conditions to outcomes, behavior, and resource use. We introduce **AlphaDiana**, a measurement system with modular interfaces for benchmarks, agents, environments, and scorers. It preserves harness-native control loops while controlling file reuse and memory retention separately. AlphaDiana supports *macro analyses of complete agents* and *micro interventions on agent components* such as tools, skills, and memory under matched conditions. We study three open-source models and three open-source harnesses on five answer-based benchmarks, with two models also tested on two interactive benchmarks. We find that harnesses *do not consistently improve the accuracy of reasoning*: gains on questions unsolved by the model alone can be outweighed by losses on questions it answers correctly without tools. This inconsistency extends to component interventions: neither skill guidance nor retained history reliably improves accuracy. Behavior and cost must also be interpreted alongside correctness: *low token entropy can accompany wrong answers*, and less output per execution can mean *more output tokens per correct answer* when accuracy falls. AlphaDiana links answer correctness and execution status to traces and resource records for comparing agents and diagnosing failures under shared task and resource policies.

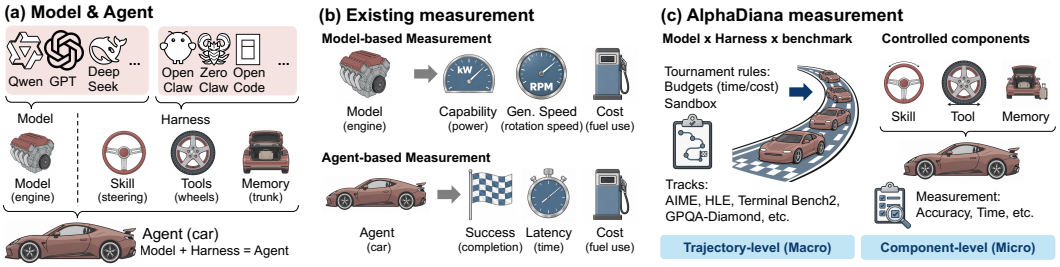


Fig. 1: **Position of AlphaDiana.** Models and agents are analogous to engines and cars. Model-centered evaluations vary models while fixing the harness. Like a tournament of cars, AlphaDiana compares complete model–harness systems under common rules through trajectory analysis, component interventions, and measurements of task accuracy, output-token use, and execution time.

1 INTRODUCTION

Benchmarks provide verifiable tasks (Rein et al., 2024; Jimenez et al., 2024), while evaluation frameworks (Kapoor et al., 2026; Harbor Framework Team, 2026) support agent evaluation. Explaining differences between agents poses three challenges: **(1) How should sessions be isolated or reused?** An *attempt* is one execution of an agent on a task. Sessions provide execution environments, while reuse policies control which files and memories later attempts can access. Uncontrolled reuse can expose earlier solutions and make comparisons depend on task order. **(2) How can comparisons preserve harness-native behavior?** Adapters need common inputs and outputs while retaining each harness's model calls, tool permissions, and feedback loop. **(3) How can outcomes be linked to agent behavior?** Scores must link to the actions, failures, and resource use of the same attempt. Fig. 1 frames these comparisons as a tournament of complete agents under common rules.

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

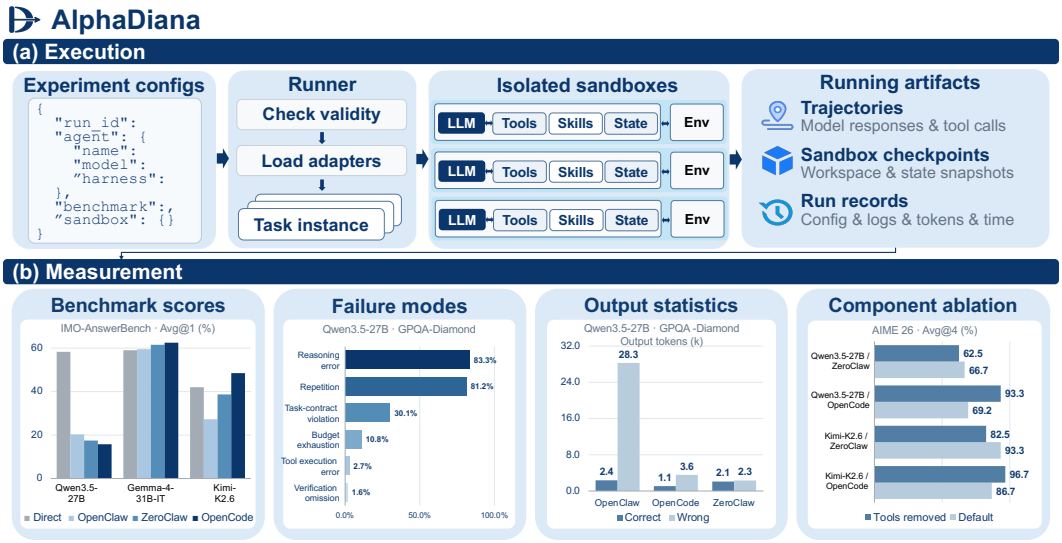


Fig. 2: **Overview of AlphaDiana.** AlphaDiana *executes* configured agents in controlled environments and saves their artifacts. *Measurement* compares performance, reasoning behavior, and costs, and tests the effects of components such as tools, skills, and memory under matched conditions.

We introduce AlphaDiana, a measurement system linking *controlled comparisons* and *component interventions* to outcomes, traces, and resource use. Researchers compare each question's result against a reference and inspect the actions and output tokens behind gains and losses. Matched interventions vary tools, skills, or memory while fixing the model, tasks, scoring, and decoding.

AlphaDiana's *modular interfaces* let users configure, combine, and extend benchmarks, agents, environments, and scorers without changing the runner. Three contributions address these challenges:

- **Session isolation and controlled reuse.** Sessions provide isolated or shared execution environments, with file resets and memory retention controlled separately. Attempts updating shared memory run sequentially without automatic retries, because failures can change what later attempts inherit.
- **Adapters preserving harness-native execution.** Adapters standardize task inputs and outputs while preserving each harness's model calls, permissions, feedback, and context updates.
- **Results linked to execution evidence.** Each recorded task execution links its score and execution status to traces and resource use. Reports include failed and unscored results and use the latest record for each requested evaluation, so retries do not count as additional evaluations of that task.

We pair the models Qwen, Gemma, and Kimi with the harnesses OpenClaw, ZeroClaw, and OpenCode on five answer-based benchmarks that score final answers. For each model, *Direct* provides the reference: one response per attempt without tools. We also test Qwen and Gemma on two interactive benchmarks, using task verifiers to check completion after file edits or commands. We find:

- **Harnesses can solve extra questions yet lower overall accuracy.** For Qwen on GPQA-Diamond, each harness solves 8–13 questions unsolved by *Direct* but fails on 18–36 that *Direct* solves. Across the four systems, 175/198 questions are solved at least once, versus 159/198 for *Direct* alone.
- **Reasoning patterns alone do not show whether an answer is correct.** Similar tool-use shares, consecutive checks, and low entropy occur with correct and incorrect answers. In annotated incorrect HLE-Verifiable attempts combining the three models, the shares labeled for repetition are 36.8% for OpenCode and 39.4% for ZeroClaw, but budget-exhaustion labels occur in 13.0% versus 0%.
- **Tools, skills, and memory do not consistently improve answers.** With four attempts per AIME 26 problem, removing tool descriptions and related instructions from Kimi/OpenCode raises correct attempts from 104 to 116 but reduces problems solved at least once from 30 to 29. Skill loading improves accuracy in only 2 of 16 comparisons. For Qwen on AIME 26, memory collected on AIME 25 and kept fixed helps OpenCode but hurts OpenClaw and ZeroClaw relative to defaults.
- **Small token savings can come with far fewer correct answers.** On common Gemma/HLE-Verifiable tasks, OpenCode uses 0.6% fewer output tokens than *Direct*, but gives 140 correct answers versus 164. We divide output from all attempts by the number of correct answers. This average increases by 16.5%. OpenClaw's highest-token 10% of attempts use 62.2% of output without a correct answer. Traces can reveal repetition, tool failures, or incorrect reasoning.

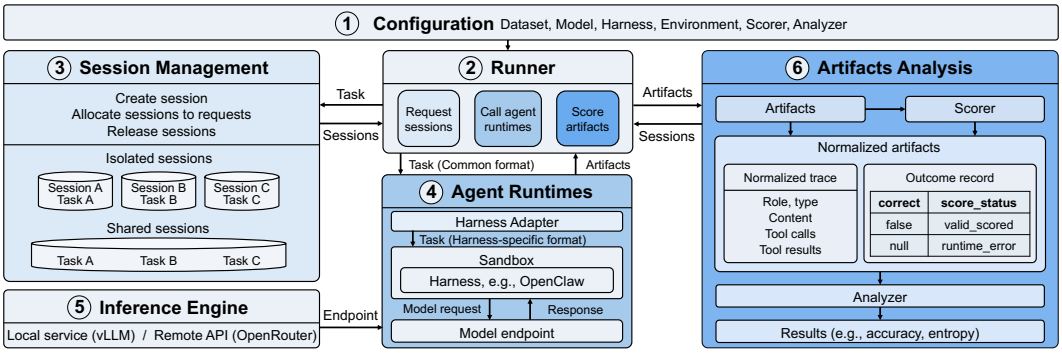


Fig. 3: **System design of AlphaDiana.** Configuration selects tasks, agents, environments, and scorers. The runner requests sessions and passes tasks to harness adapters. Native runtimes execute permitted tools in the assigned environments and query local or remote model endpoints configured by the inference engine. The runner saves available execution artifacts and scorer judgments before releasing sessions. Normalized traces and outcome records link each attempt's evidence so the analyzer can compare correctness, behavior, and resource use across tasks and inspect individual failures.

2 AlphaDiana: System Design and Interfaces

The six modules in Fig. 3 connect configuration, harness-native execution, and evidence analysis:

(1) Configuration. Configurations select tasks, model–harness pairs, environments, scorers, analyses, resource limits, evaluations per task, parallelism, and memory scope. Matched comparisons fix the model, tasks, scoring, and decoding while varying harnesses or components. The adapters in Appendix A add compatible tasks and harnesses while reusing the runner and result format.

(2) Runner. The runner schedules each requested task evaluation as a *sample*, coordinates its execution and scoring, and saves its results. A retry retains the sample identifier, linking available outputs, errors, and termination details to the same evaluation even when execution or scoring fails.

(3) Session Management. The session manager creates and releases sessions, allocating them to attempts under two policies. Adapters control file resets and memory retention separately:

- *Isolated sessions.* Where supported, a pool assigns separate sessions to concurrent attempts and resets them before reuse. Strict isolation stops the run if sandbox setup or pool preparation fails.
- *Shared sessions.* Attempts may reuse a session within one task or across tasks. Within-task sharing replaces the session and reinitializes the agent before the next task. Shared attempts run sequentially. Attempts updating shared memory are not retried automatically: a failed attempt may already have changed the shared memory that a retry or any later attempt would inherit.

(4) Agent Runtimes. Adapters translate tasks into harness-specific inputs and collect available answers, trajectories, termination reasons, and artifacts. Harnesses control model calls, action parsing, tool permissions, feedback, context updates, skills, and memory. Accepted actions invoke tools, while rejections follow harness-native feedback or termination paths. Runtimes may be local, containerized, or remote. In the matched Qwen/HLE-Verifiable example in Appendix A.3.1, OpenClaw recovers from a computation error, ZeroClaw proceeds after a shell-policy rejection, and OpenCode incorporates shell results into its context. They reach the same answer through different feedback paths. Adapters preserve these paths while translating task inputs and normalizing outputs.

(5) Inference Engine. Harnesses query local services such as vLLM or remote providers such as OpenRouter independently of the environments executing tools. Where supported, AlphaDiana records model requests, responses, and token probabilities for offline analysis of generation.

(6) Artifacts Analysis. A *normalized trace* stores available actions and feedback in a common format linked to source records. An *outcome record* stores scorer judgments separately from execution status, distinguishing scored wrong answers, execution failures, and missing scores. Reports include failed and unscored results and select the latest record per task and sample, so retries do not add samples. The analyzer compares output-token use and the proportions and ordering of reasoning and tool-use steps across outcomes. Entropy uses renormalized top-K next-token probabilities to measure concentration among the recorded candidate tokens, rather than across the model's full vocabulary.

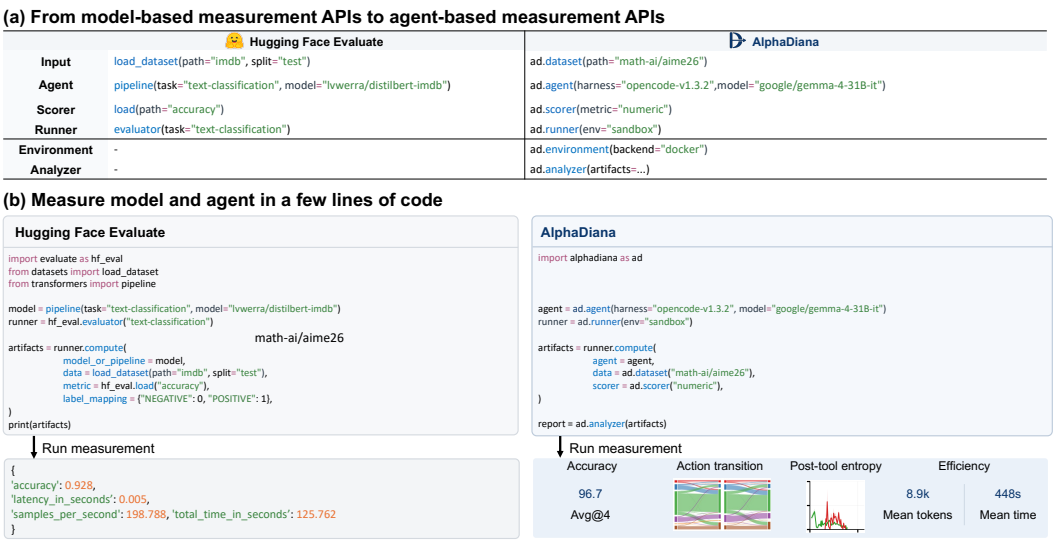


Fig. 4: **Functional interface of AlphaDiana.** (a) Evaluation abstractions in Hugging Face Evaluate and AlphaDiana. (b) Saved artifacts support performance reporting and offline process analysis.

Unified measurement interface. AlphaDiana provides composable execution and measurement interfaces. Its dataset, agent, environment, scorer, runner, and analyzer APIs support macro comparisons and matched micro interventions on tools, skills, and memory. Fig. 4 shows one composition: execution and scoring produce artifacts for analyzing accuracy, behavior, and cost. New summaries reuse these records, while component changes require new executions under matched settings.

3 Macro View: Open Agents on Verifiable Reasoning Tasks

We compare which problems complete agents solve, then examine error labels, functional steps, token entropy, and resource use to identify gains and losses that aggregate accuracy alone can conceal.

Benchmark setup. We pair Qwen3.5-27B, Gemma-4-31B-IT, and Kimi-K2.6 with OpenClaw, ZeroClaw, and OpenCode. The answer-based benchmarks score final answers against references: AIME 26, IMO-AnswerBench, GPQA-Diamond (Rein et al., 2024), HLE-Verifiable (Phan et al., 2025), and MMMU-Pro (Yue et al., 2024). Their baseline is *Direct*. For interactive tasks, we compare Qwen and Gemma with SWE-Agent on SWE Mini, a 50-task SWE-bench Verified subset (Jimenez et al., 2024), and with terminus-2 on Terminal-Bench 2 (Merrill et al., 2026). We use each harness's default configuration, without adding skill libraries or instructions to use memory. We report single-attempt accuracy (Avg@1), except on AIME 26, where Avg@4 averages correctness over four attempts per problem and Pass@4 gives the fraction of problems solved at least once. Interactive accuracy is the fraction of attempts with verified task completion, counting execution or evaluation errors as failures. A shared output limit per model call does not equalize total token use: harnesses make different numbers of calls. Appendices B.1 and C.5 specify decoding settings and budgets.

Agent performance. In Fig. 5, all three harnesses lower Qwen's scores on all five answer-based benchmarks, whereas ZeroClaw improves Gemma on all five. Effects also reverse across tasks for the same model: OpenClaw helps Kimi on HLE-Verifiable but hurts it on GPQA-Diamond. No tested harness exceeds the task-native reference on either interactive benchmark. Yet lower scores conceal additional solutions. On Qwen/GPQA-Diamond, each harness solves 8–13 questions missed by *Direct* while losing 18–36 that *Direct* solves (Tab. 6). Together, the four systems solve 175/198 questions, versus *Direct*'s 159/198. Exploiting these complementary solutions requires choosing which agent to trust on each question, a selection step requiring tests of accuracy and token cost.

Finding 1 (Fig. 5). Harnesses can add new solutions while reducing overall accuracy.

Error patterns. Fig. 6 summarizes model-assigned error labels for 2,857 incorrect attempts across three models, four systems, and four benchmarks. Labels overlap. Unscored attempts are excluded. The *repetition rate* is the fraction labeled for repeated reasoning or actions, counting each attempt

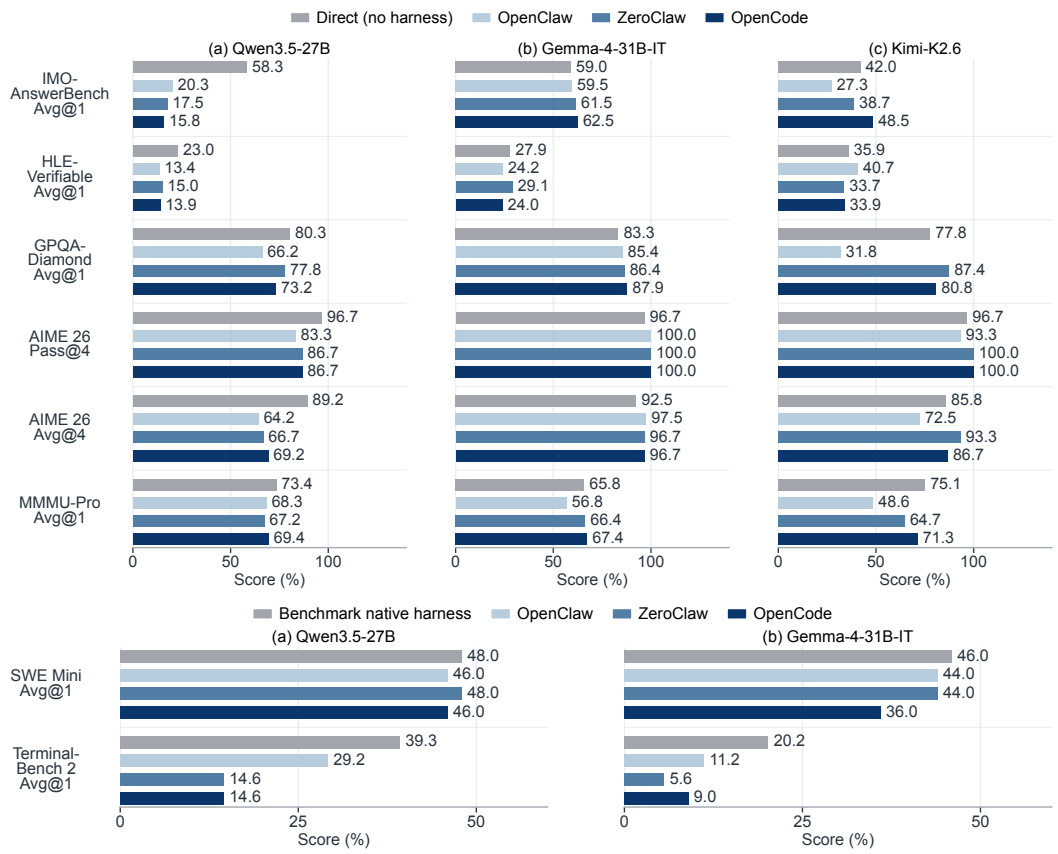


Fig. 5: **Harness effects vary across models and tasks** (Finding 1). References are *Direct* for answer-based tasks (top), and SWE-Agent and terminus-2 for the two interactive benchmarks (bottom).

once. On HLE-Verifiable, OpenCode and ZeroClaw have similar rates (36.8% and 39.4%), yet OpenCode's rate is lower within every model. Different model proportions narrow the gap.

Repetition does not measure resource consumption: budget-exhaustion labels appear in 13.0% of OpenCode's annotated incorrect HLE attempts and none of ZeroClaw's. Linking the two requires content, resource, and termination evidence. Appendix C.2 defines the labels and procedure. In the study in Appendix C.2.1, the error-labeling baseline averages 88.0% agreement with two human annotators across 17 error codes on 42 screened trajectories, versus 86.8% between the two raters.

Finding 2 (Fig. 6). Similar overall repetition rates can hide differences within every model.

Functional-step patterns. Fig. 7 compares 2,751 scored Gemma and Kimi trajectories on IMO-AnswerBench, GPQA-Diamond, and HLE-Verifiable. GPT-5.6-sol labels each *functional step*, a saved text segment or logged call (Appendix C.3). On the human calibration sample, agreement with majority judgments is 90.9% (Appendix C.3.1). Panel (a) reports step shares within each model–benchmark–system–outcome group, giving longer trajectories more weight. For incorrect Gemma/ZeroClaw trajectories on HLE-Verifiable, the 61.0% Tool Use share comes entirely from one of 75 trajectories. Computing each trajectory's share first and then averaging gives only 1.3%. Thus the large share reflects one long run, not typical behavior. Kimi/OpenCode's HLE tool-use shares are similar across outcomes. In Fig. 7(b), OpenClaw has consecutive Verification steps in both outcomes. The labels cannot distinguish successful error correction from repeated failed checks: this requires inspecting what each check found and whether the model used the result to correct its reasoning.

Finding 3 (Fig. 7). High tool-use shares can come from a single long trajectory.

Token-entropy patterns. We compute entropy from recorded top-K next-token probabilities, renormalized to sum to one, for Qwen and Gemma on AIME 26, GPQA-Diamond, and HLE-Verifiable. Lower values mean probability is concentrated on fewer recorded candidates. In Fig. 8(a), entropy often falls after tool feedback in trajectories with either correct or incorrect final answers. Fig. 8(b) also shows long incorrect *Direct* and OpenCode trajectories with low mean entropy. Thus

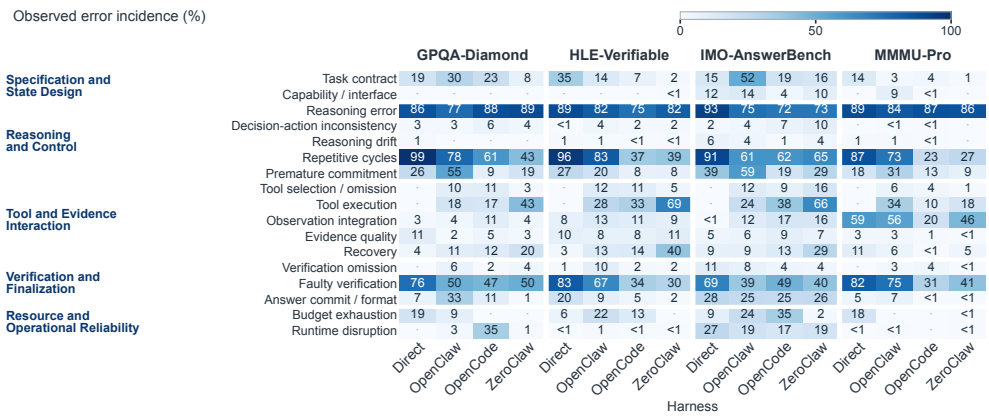


Fig. 6: **Error patterns across models and harnesses** (Finding 2). Columns pool three models; cells give error-label percentages among incorrect attempts (2,857 overall). Labels overlap; dot: zero, < 1: positive rate below 1%. Full definitions and boundaries: Appendix C.2.



Fig. 7: **Functional-step patterns** (Finding 3). (a) Step shares for correct (✓) and incorrect (×) trajectories. (b) HLE-Verifiable adjacent-label transitions, normalized per panel. Label definitions: Tab. 14. IMO/HLE denote IMO-AnswerBench/HLE-Verifiable.

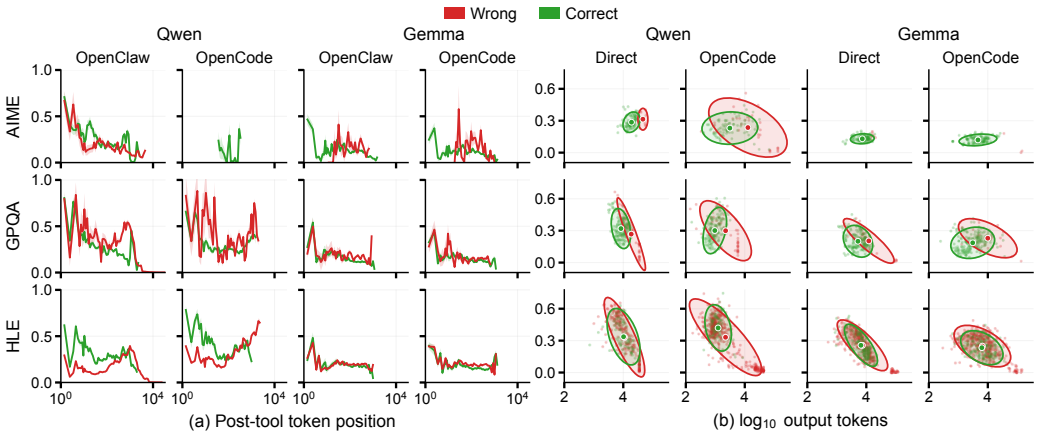


Fig. 8: **Token entropy at two scales** (Finding 4). (a) Mean entropy by post-tool token position on a logarithmic x-axis, with standard-error bands. (b) Mean entropy versus $\log_{10}$ output tokens for scored trajectories. Points represent individual trajectories. Large markers are centroids and ellipses are two-standard-deviation covariance contours.

both post-tool and whole-trajectory views show that concentrated predictions can accompany incorrect solutions. An entropy drop alone cannot establish that tool feedback has corrected the reasoning.

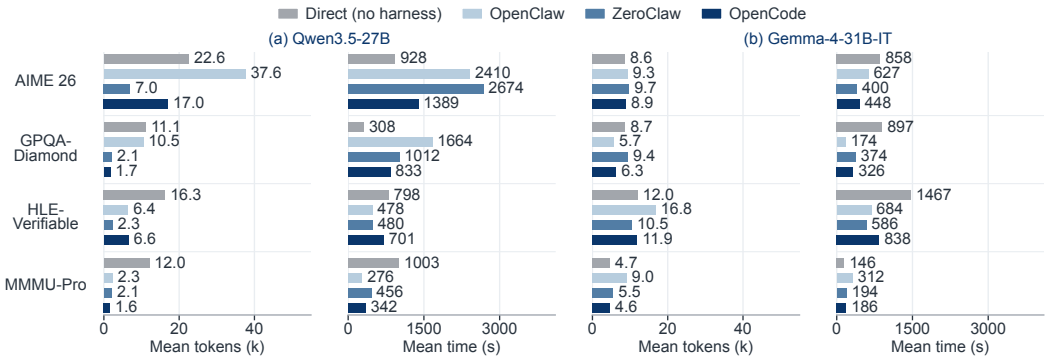


Fig. 9: **Output tokens and execution time per attempt** (Finding 5). (a) Qwen3.5-27B. (b) Gemma-4-31B-IT. Bars show mean recorded output tokens (thousands, excluding input) and execution time (seconds), including unsuccessful attempts. Record coverage can differ across systems and metrics.

Finding 4 (Fig. 8). Falling entropy after tool feedback can precede a wrong answer.

Token use and execution time. Fig. 9 compares mean output tokens and recorded execution time per attempt, including unsuccessful attempts. On Qwen/AIME 26, OpenCode uses 24.6% fewer output tokens than *Direct* but takes 49.6% longer. The reverse occurs for Gemma/ZeroClaw on GPQA-Diamond: more output accompanies shorter execution. Both comparisons have complete token records, covering 120 and 198 attempts per system, respectively. Output length measures generated text, while elapsed time includes serving, tool execution, and scheduling. These recorded times do not isolate harness overhead, but the opposite rankings show why token and time budgets need separate evaluation. Appendix B.2 documents record coverage and timing conditions.

Finding 5 (Fig. 9). Token and time measurements can rank the same systems differently.

4 Micro View: How Harness Capabilities Contribute to Reasoning

We change tools, skills, and memory, then examine the token cost of complete agents. Tool and skill interventions use Qwen and Kimi with ZeroClaw and OpenCode on GPQA-Diamond and AIME 26. Each comparison retains the model, tasks, scoring, decoding, budgets, and harness-native loop of *Default Harness*. Appendix D specifies the configuration changes and component contents.

Tool removal. *Tool Registry Removed* suppresses model-visible tool descriptions and related instructions while preserving the harness loop. Fig. 10 shows average accuracy falling for ZeroClaw in three of four model–task pairs but rising for OpenCode in three. A simpler action interface can thus improve accuracy. This comparison measures the combined change in prompting and tool availability.

On AIME 26, Kimi/OpenCode solves 29 problems on all four attempts after tool removal: 116 correct attempts, versus 104 for the default. Yet the default solves all 30 problems at least once. Higher Avg@4 thus reflects repeated success on fewer distinct problems, despite a lower Pass@4 score.

Finding 6 (Fig. 10). Tool removal can raise average accuracy while solving fewer problems.

Skill loading. *Skill Loaded* makes a math-specific or general procedural library available through the harness's native interface. Fig. 11 shows average-accuracy gains in only two of sixteen comparisons, both with Qwen. Failure to open the library cannot explain every decline: Kimi/ZeroClaw opens Math's top-level `SKILL.md` on every GPQA-Diamond attempt, yet accuracy falls. Opening the file does not show whether its advice fits the problem or is used correctly. Reading rates in Fig. 12 also depend on instructions: OpenCode uses matched instructions across libraries, whereas ZeroClaw requires reading Math but makes General optional. ZeroClaw's rates do not show library preference.

Finding 7 (Figs. 11 and 12). Skills can be read on every attempt while accuracy falls.

Memory persistence. We evaluate Qwen on AIME 26 with fixed model weights. *Cross-Rollout* keeps memory across four attempts on the same task, then clears it. *Cross-Task* keeps memory across tasks within each of four separate runs. *Cross-Dataset* uses frozen AIME 25 memory without updates. The default clears memory between attempts. We vary retention and add memory-use instructions. Raw histories and task files reset between attempts. Appendix D.4 details storage and retrieval.

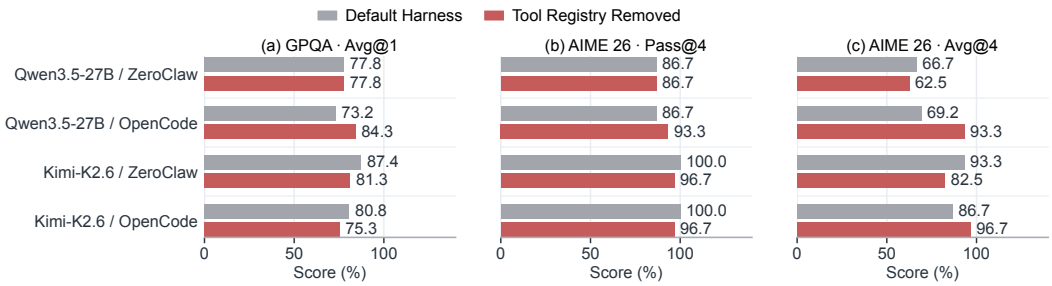


Fig. 10: **Tool removal affects harnesses differently** (Finding 6). Model-visible tool descriptions and related instructions are removed from *Default Harness*. GPQA denotes GPQA-Diamond.

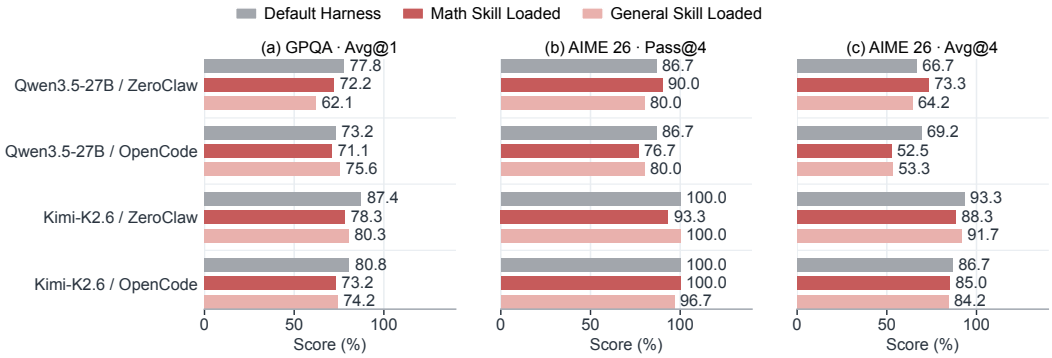


Fig. 11: **Skill loading rarely improves average accuracy** (Finding 7). Math and General denote the math-specific and general procedural skill libraries used here. GPQA denotes GPQA-Diamond.

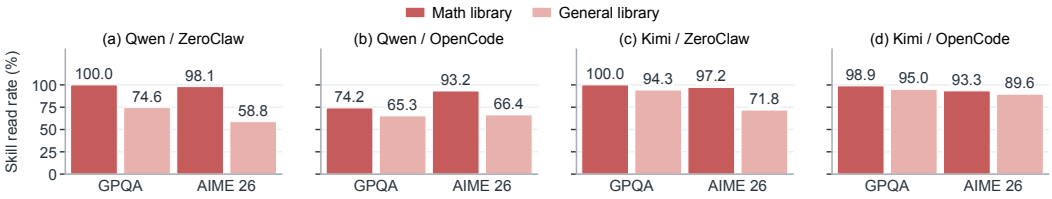


Fig. 12: **Skill-reading rates** (Finding 7): the fraction of trajectories opening the library's top-level `SKILL.md`. Opening this file alone does not establish that the advice is applied correctly.

Fig. 13 shows memory helping OpenCode and hurting OpenClaw and ZeroClaw in all three scopes. These directions persist with frozen AIME 25 memory, so gains and losses do not require updates on the target tasks. Yet none of these nine configurations surpasses the model-only *Direct* reference.

OpenCode uses prior solutions in prompts, whereas OpenClaw and ZeroClaw retrieve summaries. Retrieval, prompt length, and control loops vary, so memory format is not isolated. Fig. 14(d) shows accuracy plateauing or falling across later attempts, which can reuse prior solutions and are not independent. Cross-Task accuracy averages all tasks seen, so it also varies with task difficulty.

Finding 8 (Figs. 13 and 14). Frozen memory can help one harness and hurt another.

Output-token cost. We compare complete Gemma agents on 198 GPQA-Diamond and 587 HLE-Verifiable tasks with token records for all systems, following Appendix B.2. We report total output divided by either all attempts (*tokens per attempt*) or correct attempts (*cost per success*, labeled tokens per solved attempt). Both include output from incorrect answers, unscored attempts, and execution failures, but exclude inputs. Cost per success thus equals tokens per attempt divided by accuracy as a fraction. This differs from averaging output tokens only over successful attempts.

In Fig. 15, OpenCode saves 0.6% of output tokens per HLE-Verifiable attempt relative to *Direct*, yet its cost per success is 16.5% higher. Nearly unchanged output is spread across fewer correct attempts. When accuracy falls, token savings reduce cost per success only if their relative reduction exceeds the relative accuracy loss. A ranking based only on output length reverses this cost comparison.

Finding 9 (Fig. 15). Lower token use per attempt can mean higher cost per success.

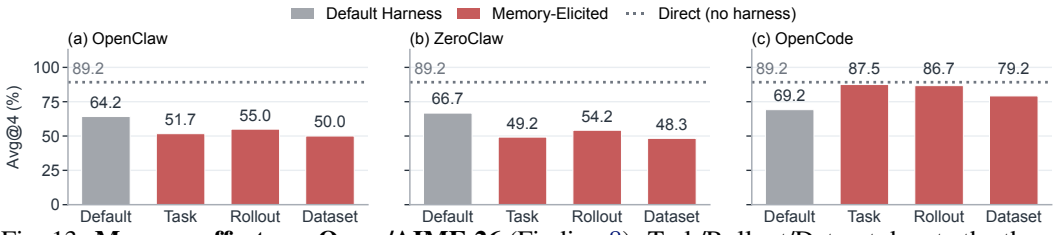


Fig. 13: **Memory effects on Qwen/AIME 26** (Finding 8). Task/Rollout/Dataset denote the three persistence scopes. The dotted line marks *Direct*'s 89.2% Avg@4, without a harness or tools.

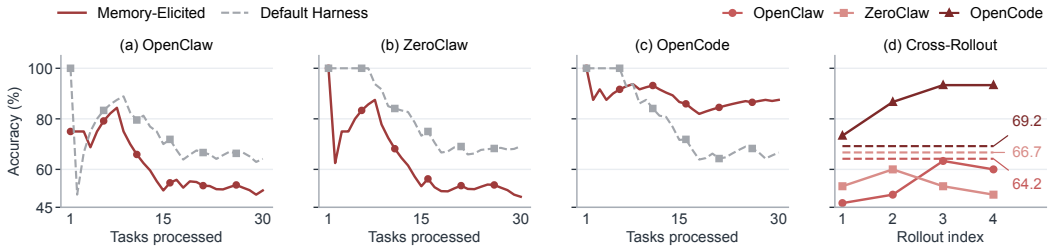


Fig. 14: **Accuracy as memory accumulates** (Finding 8). (a–c) Cross-Task running accuracy from separate sequential runs, with default scores that may differ from Fig. 13. (d) Cross-Rollout accuracy.

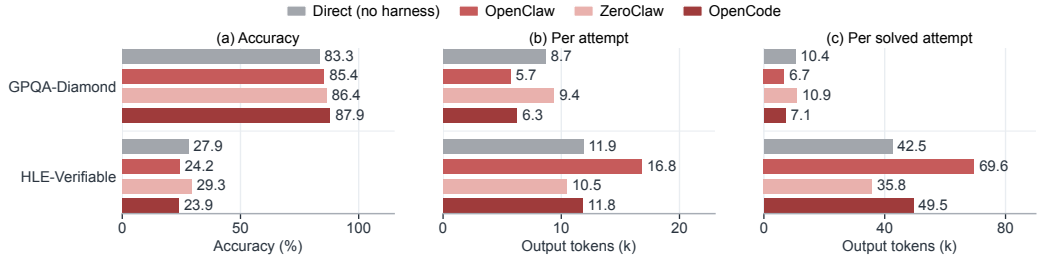


Fig. 15: **Output-token cost of success** (Finding 9). Costs include unsuccessful attempts.

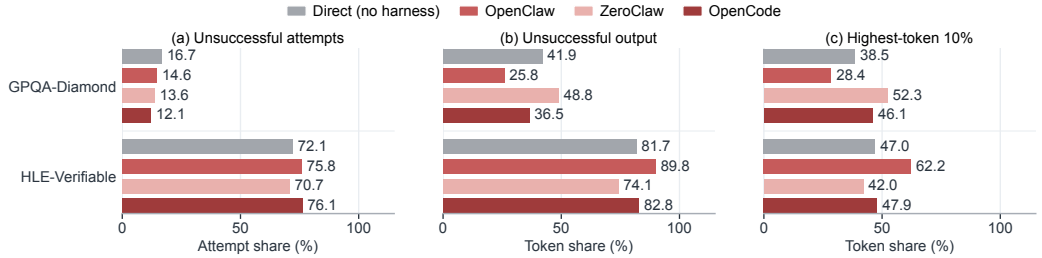


Fig. 16: **Where output tokens are spent** (Finding 10). Same tasks as Fig. 15. Panels show the unsuccessful-attempt fraction, its output share, and the highest-token 10%'s output share.

Output concentration. Figs. 15 and 16 show why averaging only successful attempts understates cost. On HLE-Verifiable, OpenClaw's correct attempts average fewer tokens than *Direct*'s, yet its cost per success is 63.9% higher because unsuccessful attempts consume 89.8% of its output. Its highest-token 10% consume 62.2% without a correct answer. These totals do not reveal when failure becomes predictable. Stopping rules must be tested for both accuracy and token use on matched tasks. The highest-token and unsuccessful groups overlap, so their output shares cannot be added.

Finding 10 (Fig. 16). 10% of attempts can consume most output without a correct answer.

5 Conclusion

AlphaDiana links harness-native execution and controlled session reuse to outcomes, traces, and resource records. Harnesses can solve questions missed by the model alone yet reduce accuracy, while shorter output per attempt can increase cost per success. Component changes should be judged by which answers become correct or incorrect and total output per correct answer. Matched interventions and saved traces let researchers test these changes and inspect failures in individual attempts.

AI USE STATEMENT

We used generative AI tools to polish the manuscript and `gpt-5.6-sol` to label functional steps for Fig. 7 and error categories for Fig. 6. The annotation procedures and evidence limitations are described in Appendices C.3 and C.2. The authors are responsible for all content, including AI-assisted text, analyses, and conclusions.

ETHICS STATEMENT

AlphaDiana aims to make agent measurement more rigorous, reproducible, and interpretable by making the model-harness boundary explicit and preserving auditable execution traces, which can help avoid misleading leaderboard-style comparisons and diagnose harness-induced failures. Stronger evaluation infrastructure may also accelerate the development of more capable autonomous agents, and trajectory logging can raise privacy and data-governance concerns when evaluations involve sensitive prompts, files, or tool outputs. We therefore encourage sandboxed execution, access control, privacy-preserving logging, and transparent reporting of evaluation scope and limitations.

REPRODUCIBILITY STATEMENT

To support reproducibility, Sec. 2 and Appendix A describe the evaluation workflow, module interfaces, execution-state management, and saved result records. Appendix B documents benchmarks, scorers, harness versions, sampling and decoding settings, measurement coverage, and computing resources. Appendix C details the error annotation, functional-step labeling and aggregation, token-entropy analysis, and runtime limits. Appendix D specifies the tool, skill, and memory interventions. Together, these descriptions identify the experimental conditions and analysis conventions needed to reproduce the reported comparisons.

REFERENCES

Reyna Abhyankar, Qi Qi, and Yiying Zhang. Osworld-human: Benchmarking the efficiency of computer-use agents. *arXiv preprint arXiv:2506.16042*, 2025. URL https://arxiv.org/abs/2506.16042.

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025. URL https://arxiv.org/abs/2507.19457.

Mohammad Almansoori, Komal Kumar, and Hisham Cholakkal. Medagentsim: Self-evolving multi-agent simulations for realistic clinical interactions. In *MICCAI*, 2025. URL https://papers.miccai.org/miccai-2025/0537-Paper2575.html.

Adam AlSayyad, Kelvin Yuxiang Huang, and Richik Pal. Agenttrace: A structured logging framework for agent system observability. *arXiv preprint arXiv:2602.10133*, 2026. URL https://arxiv.org/abs/2602.10133.

Kinjal Basu, Ibrahim Abdelaziz, Kiran Kate, Mayank Agarwal, Maxwell Crouse, Yara Rizk, Kelsey Bradford, Asim Munawar, Sadhana Kumaravel, Saurabh Goyal, Xin Wang, Luis A. Lastras, and Pavan Kapanipathi. Nestful: A benchmark for evaluating llms on nested sequences of api calls. *arXiv preprint arXiv:2409.03797*, 2024. URL https://arxiv.org/abs/2409.03797.

Varun Pratap Bhardwaj. Agentassay: Token-efficient regression testing for non-deterministic ai agent workflows. *arXiv preprint arXiv:2603.02601*, 2026. URL https://arxiv.org/abs/2603.02601.

Bjarni Haukur Bjarnason, André Silva, and Martin Monperrus. On randomness in agentic evals. In *ICLR Workshop on Agents in the Wild*, 2026. URL https://openreview.net/forum?id=UFukKhqCpr.

Nghi D. Q. Bui. Building effective ai coding agents for the terminal: Scaffolding, harness, context engineering, and lessons learned. *arXiv preprint arXiv:2603.05344*, 2026. URL https://arxiv.org/abs/2603.05344.

Christoph Bühler, Matteo Biagiola, Luca Di Grazia, and Guido Salvaneschi. Securing ai agent execution. *arXiv preprint arXiv:2510.21236*, 2025. URL https://arxiv.org/abs/2510.21236v1.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde 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, Josh 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. *arXiv preprint arXiv:2107.03374*, 2021. URL https://arxiv.org/abs/2107.03374.

Zehui Chen, Weihua Du, Wenwei Zhang, Kuikun Liu, Jiangning Liu, Miao Zheng, Jingming Zhuo, Songyang Zhang, Dahua Lin, Kai Chen, and Feng Zhao. T-eval: Evaluating the tool utilization capability of large language models step by step. In *ACL*, 2024. URL https://aclanthology.org/2024.acl-long.515/.

Daixuan Cheng, Shaohan Huang, Yuxian Gu, Huatong Song, Guoxin Chen, Li Dong, Wayne Xin Zhao, Ji-Rong Wen, and Furu Wei. Computer environments elicit general agentic intelligence in llms. *arXiv preprint arXiv:2601.16206*, 2026. URL https://arxiv.org/abs/2601.16206.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025. URL https://arxiv.org/abs/2504.19413.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021. URL https://arxiv.org/abs/2110.14168.

João Coelho, Jingjie Ning, Jingyuan He, Kangrui Mao, Abhijay Paladugu, Pranav Setlur, Jiahe Jin, Jamie Callan, João Magalhães, Bruno Martins, and Chenyan Xiong. Deepresearchgym: A free, transparent, and reproducible evaluation sandbox for deep research. *arXiv preprint arXiv:2505.19253*, 2025. URL https://arxiv.org/abs/2505.19253.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025. URL https://arxiv.org/abs/2501.12948.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. In *NeurIPS*, 2023. URL https://arxiv.org/abs/2306.06070.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. Workarena: How capable are web agents at solving common knowledge work tasks? In *ICML*, 2024. URL https://proceedings.mlr.press/v235/drouin24a.html.

Mingxuan Du, Benfeng Xu, Chiwei Zhu, Shaohan Wang, Pengyu Wang, Xiaorui Wang, and Zhendong Mao. A-rag: Scaling agentic retrieval-augmented generation via hierarchical retrieval interfaces. *arXiv preprint arXiv:2602.03442*, 2026. URL https://arxiv.org/abs/2602.03442.

Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. Plan-and-act: Improving planning of agents for long-horizon tasks. In *ICML*, 2025. URL https://openreview.net/forum?id=ybA4EcMmUZ.

Tongtong Feng, Xin Wang, Zekai Zhou, Ren Wang, Yuwei Zhan, Guangyao Li, Qing Li, and Wenwu Zhu. Evoagent: Self-evolving agent with continual world model for long-horizon tasks. *arXiv preprint arXiv:2502.05907*, 2025. URL https://arxiv.org/abs/2502.05907v2.

Mohamed Amine Ferrag, Norbert Tihanyi, and Merouane Debbah. From llm reasoning to autonomous ai agents: A comprehensive review. *arXiv preprint arXiv:2504.19678*, 2025. URL https://arxiv.org/abs/2504.19678.

Yudong Gao, Zongjie Li, Yuanyuan Yuan, Zimo Ji, Pingchuan Ma, and Shuai Wang. Skillreducer: Optimizing llm agent skills for token efficiency. *arXiv preprint arXiv:2603.29919*, 2026. URL https://arxiv.org/abs/2603.29919.

Aniketh Garikaparthi, Manasi Patwardhan, and Arman Cohan. Researchgym: Evaluating language model agents on real-world ai research. In *ICLR Workshop on Agents in the Wild*, 2026. URL https://openreview.net/forum?id=eY9RXNbIwr.

Elliot Glazer, Ege Erdil, Tamay Besiroglu, Diego Chicharro, Evan Chen, Alex Gunning, Caroline Falkman Olsson, Jean-Stanislas Denain, Anson Ho, Emily de Oliveira Santos, Olli Järviniemi, Matthew Barnett, Robert Sandler, Matej Vrzala, Jaime Sevilla, Qiuyu Ren, Elizabeth Pratt, Lionel Levine, Grant Barkley, Natalie Stewart, Bogdan Grechuk, Tetiana Grechuk, Shreepranav Varma Enugandla, and Mark Wildon. Frontiermath: A benchmark for evaluating advanced mathematical reasoning in AI. *arXiv preprint arXiv:2411.04872*, 2024. URL https://arxiv.org/abs/2411.04872.

Google. Gemini thinking. URL https://ai.google.dev/gemini-api/docs/thinking. Online documentation; accessed 2026-09-16.

Xiyu Guo, Shan Wang, Chunfang Ji, Xuefeng Zhao, Wenhao Xi, Yaoyao Liu, Qinglan Li, Chao Deng, and Junlan Feng. Towards generalized routing: Model and agent orchestration for adaptive and efficient inference. *arXiv preprint arXiv:2509.07571*, 2025. URL https://arxiv.org/abs/2509.07571.

Bernal Jiménez Gutiérrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. Hipporag: Neurobiologically inspired long-term memory for large language models. In *NeurIPS*, 2024. URL https://openreview.net/forum?id=hkujvAPVsg.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments, 2026. URL https://github.com/harbor-framework/harbor.

Chaoyue He, Xin Zhou, Di Wang, Hong Xu, Wei Liu, and Chunyan Miao. Harness engineering for language agents: The harness layer as control, agency, and runtime. *Preprints*, 2026. URL https://www.preprints.org/manuscript/202603.1756.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. Webvoyager: Building an end-to-end web agent with large multimodal models. *arXiv preprint arXiv:2401.13919*, 2024. URL https://arxiv.org/abs/2401.13919.

Yufei He, Ruoyu Li, Alex Chen, Yue Liu, Yulin Chen, Yuan Sui, Cheng Chen, Yi Zhu, Luca Luo, Frank Yang, and Bryan Hooi. Enabling self-improving agents to learn at test time with human-in-the-loop guidance. In *EMNLP 2025 Industry Track*, 2025. URL https://aclanthology.org/2025.emnlp-industry.115/.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. *arXiv preprint arXiv:2103.03874*, 2021. URL https://arxiv.org/abs/2103.03874.

Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. Hiagent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. In *ACL*, 2025. URL https://aclanthology.org/2025.acl-long.1575/.

Yuanzhe Hu, Yu Wang, and Julian McAuley. Evaluating memory in llm agents via incremental multi-turn interactions. In *ICLR*, 2026. URL https://openreview.net/forum?id=DT7JyQC3MR.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024. URL https://arxiv.org/abs/2403.07974.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *arXiv preprint arXiv:2504.07164*, 2025. URL https://arxiv.org/abs/2504.07164.

Smriti Jha, Matteo Paltenghi, Chandra Maddila, Vijayaraghavan Murali, Shubham Ugare, and Satish Chandra. Prodcodebench: A production-derived benchmark for evaluating ai coding agents. *arXiv preprint arXiv:2604.01527*, 2026. URL https://arxiv.org/abs/2604.01527v1.

Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression. In *ACL*, 2024. URL https://aclanthology.org/2024.acl-long.91/.

Zhengbo Jiao, Shaobo Wang, Zifan Zhang, Xuan Ren, Wei Wang, Bing Zhao, Hu Wei, and Linfeng Zhang. Agentic proposing: Enhancing large language model reasoning via compositional skill synthesis. *arXiv preprint arXiv:2602.03279*, 2026. URL https://arxiv.org/abs/2602.03279.

Cheng Jiayang, Dongyu Ru, Lin Qiu, Yiyang Li, Xuezhi Cao, Yangqiu Song, and Xunliang Cai. Amemgym: Interactive memory benchmarking for assistants in long-horizon conversations. In *ICLR*, 2026. URL https://openreview.net/forum?id=sfrVLzsmlf.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. Swe-bench: Can language models resolve real-world GitHub issues? In *ICLR*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A. Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. Acon: Optimizing context compression for long-horizon llm agents. *arXiv preprint arXiv:2510.00615*, 2025. URL https://arxiv.org/abs/2510.00615.

Sayash Kapoor, Benedikt Stroebl, Peter Kirgis, Nitya Nadgir, Zachary S Siegel, Boyi Wei, Tianci Xue, Ziru Chen, Felix Chen, Saiteja Utpala, Franck Ndzomga, Dheeraj Oruganty, Sophie Luskin, Kangheng Liu, Botao Yu, Amit Arora, Dongyoon Hahm, Harsh Trivedi, Huan Sun, Juyong Lee, Tengjun Jin, Yifan Mai, Yifei Zhou, Yuxuan Zhu, Rishi Bommasani, Daniel Kang, Dawn Song, Peter Henderson, Yu Su, Percy Liang, and Arvind Narayanan. Holistic agent leaderboard: The missing infrastructure for ai agent evaluation. In *ICLR*, 2026. URL https://openreview.net/forum?id=vUaY1t64ZZ.

Doyoung Kim, Zhiwei Ren, Jie Hao, Zhongkai Sun, Lichao Wang, Xiyao Ma, Zack Ye, Xu Han, Jun Yin, Heng Ji, Wei Shen, Xing Fan, Benjamin Yao, and Chenlei Guo. Beyond perfect apis: A comprehensive evaluation of llm agents under real-world api complexity. *arXiv preprint arXiv:2601.00268*, 2026. URL https://arxiv.org/abs/2601.00268.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. *arXiv preprint arXiv:2401.13649*, 2024. URL https://arxiv.org/abs/2401.13649.

Lingavasan Suresh Kumar, Yang Ba, and Rong Pan. Memarchitect: A policy driven memory governance layer. *arXiv preprint arXiv:2603.18330*, 2026. URL https://arxiv.org/abs/2603.18330.

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

Han Li, Letian Zhu, Bohan Zhang, Rili Feng, Jiaming Wang, Yue Pan, Earl T. Barr, Federica Sarro, Zhaoyang Chu, and He Ye. Contextbench: A benchmark for context retrieval in coding agents. *arXiv preprint arXiv:2602.05892*, 2026. URL https://arxiv.org/abs/2602.05892.

Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, et al. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110*, 2022. URL https://arxiv.org/abs/2211.09110.

George Ling, Shanshan Zhong, and Richard Huang. Agent skills: A data-driven analysis of claude skills for extending large language model functionality. *arXiv preprint arXiv:2602.08004*, 2026. URL https://arxiv.org/abs/2602.08004.

Hanbing Liu, Chunhao Tian, Nan An, Ziyuan Wang, Pinyan Lu, Changyuan Yu, and Qi Qi. Budget-constrained agentic large language models: Intention-based planning for costly tool use. *arXiv preprint arXiv:2602.11541*, 2026a. URL https://arxiv.org/abs/2602.11541.

Jiaqi Liu, Yaofeng Su, Peng Xia, Siwei Han, Zeyu Zheng, Cihang Xie, Mingyu Ding, and Huaxiu Yao. Simplemem: Efficient lifelong memory for llm agents. *arXiv preprint arXiv:2601.02553*, 2026b. URL https://arxiv.org/abs/2601.02553.

Shanyv Liu, Xuyang Yuan, Tao Chen, Zijun Zhan, Zhu Han, Danyang Zheng, Weishan Zhang, and Shaohua Cao. Caster: Breaking the cost-performance barrier in multi-agent orchestration via context-aware strategy for task efficient routing. *arXiv preprint arXiv:2601.19793*, 2026c. URL https://arxiv.org/abs/2601.19793.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating LLMs as agents. In *ICLR*, 2024. URL https://arxiv.org/abs/2308.03688.

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. *arXiv preprint arXiv:2603.03329*, 2026. URL https://arxiv.org/abs/2603.03329.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Felix Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities. *arXiv preprint arXiv:2408.04682*, 2024. URL https://arxiv.org/abs/2408.04682.

Miao Lu, Weiwei Sun, Weihua Du, Zhan Ling, Xuesong Yao, Kang Liu, and Jiecao Chen. Scaling llm multi-turn rl with end-to-end summarization-based context management. *arXiv preprint arXiv:2510.06727*, 2025a. URL https://arxiv.org/abs/2510.06727.

Mingfei Lu, Mengjia Wu, Feng Liu, Jiawei Xu, Weikai Li, Haoyang Wang, Zhengdong Hu, Ying Ding, Yizhou Sun, Jie Lu, and Yi Zhang. Choosing how to remember: Adaptive memory structures for llm agents. *arXiv preprint arXiv:2602.14038*, 2026. URL https://arxiv.org/abs/2602.14038.

Pan Lu, Bowen Chen, Sheng Liu, Rahul Thapa, Joseph Boen, and James Zou. Octotools: An agentic framework with extensible tools for complex reasoning. *arXiv preprint arXiv:2502.11271*, 2025b. URL https://arxiv.org/abs/2502.11271.

Xing Han Lù, Amirhossein Kazemnejad, Nicholas Meade, Arkil Patel, Dongchan Shin, Alejandra Zambrano, Karolina Stańczak, Peter Shaw, Christopher J. Pal, and Siva Reddy. Agentrewardbench: Evaluating automatic evaluations of web agent trajectories. *arXiv preprint arXiv:2504.08942*, 2025. URL https://arxiv.org/abs/2504.08942.

Siyuan Ma, Bo Gao, Xiaojun Jia, Simeng Qin, Tianlin Li, Ke Ma, Xiaoshuang Jia, Wenqi Ren, and Yang Liu. Odar: Principled adaptive routing for llm reasoning via active inference. *arXiv preprint arXiv:2602.23681*, 2026. URL https://arxiv.org/abs/2602.23681.

Wenquan Ma, Jiayan Nan, Wenlong Wu, and Yize Chen. What deserves memory: Adaptive memory distillation for llm agents. *arXiv preprint arXiv:2508.03341*, 2025. URL https://arxiv.org/abs/2508.03341.

Alexandre Cristovão Maiorano. Llm readiness harness: Evaluation, observability, and ci gates for llm/rag applications. *arXiv preprint arXiv:2603.27355*, 2026. URL https://arxiv.org/abs/2603.27355.

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

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: A benchmark for general ai assistants. *arXiv preprint arXiv:2311.12983*, 2023. URL https://arxiv.org/abs/2311.12983.

Zahra Moslemi, Keerthi Koneru, Yen-Ting Lee, Sheethal Kumar, and Ramesh Radhakrishnan. Polaris: Typed planning and governed execution for agentic ai in back-office automation. *arXiv preprint arXiv:2601.11816*, 2026. URL https://arxiv.org/abs/2601.11816.

Sriraam Natarajan, Saurabh Mathur, Sahil Sidheekh, Wolfgang Stammer, and Kristian Kersting. Human-in-the-loop or ai-in-the-loop? automate or collaborate? *arXiv preprint arXiv:2412.14232*, 2024. URL https://arxiv.org/abs/2412.14232.

Deepak Nathani, Lovish Madaan, Nicholas Roberts, Nikolay Bashlykov, Ajay Menon, Vincent Moens, Amar Budhiraja, Despoina Magka, Vladislav Vorotilov, Gaurav Chaurasia, Dieuwke Hupkes, Ricardo Silveira Cabral, Tatiana Shavrina, Jakob Foerster, Yoram Bachrach, William Yang Wang, and Roberta Raileanu. Mlgym: A new framework and benchmark for advancing ai research agents. *arXiv preprint arXiv:2502.14499*, 2025. URL https://arxiv.org/abs/2502.14499.

Zhiyu Ni, Yifeng Xiao, and Zheng Liang. Agentified assessment of logical reasoning agents. *arXiv preprint arXiv:2603.02788*, 2026. URL https://arxiv.org/abs/2603.02788.

OpenAI. Reasoning models. URL https://developers.openai.com/api/docs/guides/reasoning/. Online documentation; accessed 2026-09-16.

OpenAI. Learning to reason with LLMs, 2024. URL https://openai.com/index/learning-to-reason-with-llms/.

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

Young Bin Park. Graph-native cognitive memory for ai agents: Formal belief revision semantics for versioned memory architectures. *arXiv preprint arXiv:2603.17244*, 2026. URL https://arxiv.org/abs/2603.17244.

L. Phan et al. Humanity's last exam. *arXiv preprint arXiv:2501.14249*, 2025. URL https://arxiv.org/abs/2501.14249.

Stephan Rabanser, Sayash Kapoor, Peter Kirgis, Kangheng Liu, Saiteja Utpala, and Arvind Narayanan. Towards a science of ai agent reliability. *arXiv preprint arXiv:2602.16666*, 2026. URL https://arxiv.org/abs/2602.16666.

Preston Rasmussen, Pavlo Paliychuk, Travis Beauvais, Jack Ryan, and Daniel Chalef. Zep: A temporal knowledge graph architecture for agent memory. *arXiv preprint arXiv:2501.13956*, 2025. URL https://arxiv.org/abs/2501.13956.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A graduate-level google-proof Q&A benchmark. In *COLM*, 2024. URL https://openreview.net/forum?id=Ti67584b98.

Alexander Roman and Jacob Roman. Orchestral ai: A framework for agent orchestration. *arXiv preprint arXiv:2601.02577*, 2026. URL https://arxiv.org/abs/2601.02577.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *NeurIPS*, 2023. URL https://openreview.net/forum?id=Yacmpz84TH.

Quan Shi, Alexandra Zytek, Pedram Razavi, Karthik Narasimhan, and Victor Barres. τ-Knowledge: Evaluating conversational agents over unstructured knowledge. *arXiv preprint arXiv:2603.04370*, 2026. URL https://arxiv.org/abs/2603.04370.

Zachary S. Siegel, Sayash Kapoor, Nitya Nadgir, Benedikt Stroebl, and Arvind Narayanan. Core-bench: Fostering the credibility of published research through a computational reproducibility agent benchmark. *arXiv preprint arXiv:2409.11363*, 2024. URL https://arxiv.org/abs/2409.11363.

Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon llm agent via context-folding. *arXiv preprint arXiv:2510.11967*, 2025. URL https://arxiv.org/abs/2510.11967.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V. Le, Ed H. Chi, Denny Zhou, and Jason Wei. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*, 2022. URL https://arxiv.org/abs/2210.09261.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In *EACL*, 2026. URL https://aclanthology.org/2026.eacl-long.333/.

Aristidis Vasilopoulos. Codified context: Infrastructure for ai agents in a complex codebase. *arXiv preprint arXiv:2602.20478*, 2026. URL https://arxiv.org/abs/2602.20478.

Nikhil Verma. Active context compression: Autonomous memory management in llm agents. *arXiv preprint arXiv:2601.07190*, 2026. URL https://arxiv.org/abs/2601.07190.

Sri Vatsa Vuddanti and Satwik Kumar Chittiprolu. Recoverability has a law: The err measure for tool-augmented agents. *arXiv preprint arXiv:2601.22352*, 2026. URL https://arxiv.org/abs/2601.22352.

Guangya Wan, Mingyang Ling, Xiaoqi Ren, Rujun Han, Sheng Li, and Zizhao Zhang. Compass: Enhancing agent long-horizon reasoning with evolving context. *arXiv preprint arXiv:2510.08790*, 2025. URL https://arxiv.org/abs/2510.08790.

Xiaoxuan Wang, Ziniu Hu, Pan Lu, Yanqiao Zhu, Jieyu Zhang, Satyen Subramaniam, Arjun R. Loomba, Shichang Zhang, Yizhou Sun, and Wei Wang. Scibench: Evaluating college-level scientific problem-solving abilities of large language models. In *ICML*, 2024. URL https://proceedings.mlr.press/v235/wang24z.html.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *ICLR*, 2023. URL https://openreview.net/forum?id=1PL1NIMMrw.

Yuhang Wang, Yuling Shi, Mo Yang, Rongrui Zhang, Shilin He, Heng Lian, Yuting Chen, Siyu Ye, Kai Cai, and Xiaodong Gu. Swe-pruner: Self-adaptive context pruning for coding agents. *arXiv preprint arXiv:2601.16746*, 2026. URL https://arxiv.org/abs/2601.16746.

Zhenting Wang, Qi Chang, Hemani Patel, Shashank Biju, Cheng-En Wu, Quan Liu, Aolin Ding, Alireza Rezazadeh, Ankit Shah, Yujia Bao, and Eugene Siow. Mcp-bench: Benchmarking tool-using LLM agents with complex real-world tasks via MCP servers. *arXiv preprint arXiv:2508.20453*, 2025a. URL https://arxiv.org/abs/2508.20453.

Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. Inducing programmatic skills for agentic tasks. *arXiv preprint arXiv:2504.06821*, 2025b. URL https://arxiv.org/abs/2504.06821.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, 2022. URL https://arxiv.org/pdf/2201.11903.

Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus, and Amelia Glaese. Browsecomp: A simple yet challenging benchmark for browsing agents. *arXiv preprint arXiv:2504.12516*, 2025a. URL https://arxiv.org/abs/2504.12516.

Tianxin Wei, Noveen Sachdeva, Benjamin Coleman, Zhankui He, Yuanchen Bei, Xuying Ning, Mengting Ai, Yunzhe Li, Jingrui He, Ed H. Chi, Chi Wang, Shuo Chen, Fernando Pereira, Wang-Cheng Kang, and Derek Zhiyuan Cheng. Evo-memory: Benchmarking llm agent test-time learning with self-evolving memory. *arXiv preprint arXiv:2511.20857*, 2025b. URL https://arxiv.org/abs/2511.20857.

Jialong Wu, Wenbiao Yin, Yong Jiang, Zhenglin Wang, Zekun Xi, Runnan Fang, Linhai Zhang, Yulan He, Deyu Zhou, Pengjun Xie, and Fei Huang. Webwalker: Benchmarking llms in web traversal. *arXiv preprint arXiv:2501.07572*, 2025. URL https://arxiv.org/abs/2501.07572.

Yunze Wu, Dayuan Fu, Weiye Si, Zhen Huang, Mohan Jiang, Keyu Li, Shijie Xia, Jie Sun, Tianze Xu, Yang Xiao, Pengrui Lu, Xiaojie Cai, Lyumanshan Ye, Wenhong Zhu, Xiangkun Hu, and Pengfei Liu. Innovatorbench: Evaluating agents' ability to conduct innovative ai research. In *ICLR*, 2026. URL https://openreview.net/forum?id=w8rZ2Jd6Jo.

Yuan-An Xiao, Pengfei Gao, Chao Peng, and Yingfei Xiong. Reducing cost of llm agents with trajectory reduction. *Proceedings of the ACM on Software Engineering*, 2026. URL https://arxiv.org/abs/2509.23586.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *arXiv preprint arXiv:2404.07972*, 2024. URL https://arxiv.org/abs/2404.07972.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *arXiv preprint arXiv:2502.12110*, 2025. URL https://arxiv.org/abs/2502.12110.

B. Y. Yan, Chaofan Li, Hongjin Qian, Shuqi Lu, and Zheng Liu. General agentic memory via deep research. *arXiv preprint arXiv:2511.18423*, 2025. URL https://arxiv.org/abs/2511.18423.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025. URL https://arxiv.org/abs/2505.09388.

Shuo Yang, Soyeon Caren Han, Xueqi Ma, Yan Li, Mohammad Reza Ghasemi Madani, and Eduard Hovy. Evotool: Self-evolving tool-use policy optimization in llm agents via blame-aware mutation and diversity-aware selection. *arXiv preprint arXiv:2603.04900*, 2026a. URL https://arxiv.org/abs/2603.04900.

Yutao Yang, Junsong Li, Qianjun Pan, Bihao Zhan, Yuxuan Cai, Lin Du, Jie Zhou, Kai Chen, Qin Chen, Xin Li, Bo Zhang, and Liang He. Autoskill: Experience-driven lifelong learning via skill self-evolution. *arXiv preprint arXiv:2603.01145*, 2026b. URL https://arxiv.org/abs/2603.01145.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik R. Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *NeurIPS*, 2023a. URL https://openreview.net/forum?id=5Xc1ecxO1h.

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 *ICLR*, 2023b. URL https://openreview.net/forum?id=WE_vluYUL-X.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024. URL https://arxiv.org/abs/2406.12045.

Ori Yoran, Samuel Joseph Amouyal, Chaitanya Malaviya, Ben Bogin, Ofir Press, and Jonathan Berant. Assistantbench: Can web agents solve realistic and time-consuming tasks? In *EMNLP*, 2024. URL https://arxiv.org/abs/2407.15711.

Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. Utboost: Rigorous evaluation of coding agents on swe-bench. In *ACL*, 2025a. URL https://aclanthology.org/2025.acl-long.189/.

Zhouliang Yu, Ruotian Peng, Keyi Ding, Yizhe Li, Zhongyuan Peng, Minghao Liu, Yifan Zhang, Zheng Yuan, Huajian Xin, Wenhao Huang, Yandong Wen, Ge Zhang, and Weiyang Liu. Formalmath: Benchmarking formal mathematical reasoning of large language models. *arXiv preprint arXiv:2505.02735*, 2025b. URL https://arxiv.org/abs/2505.02735.

Xiang Yue, Tianyu Zheng, Yuansheng Ni, Yubo Wang, Kai Zhang, Shengbang Tong, Yuxuan Sun, Botao Yu, Ge Zhang, Huan Sun, Yu Su, Wenhu Chen, and Graham Neubig. MMMU-Pro: A more robust multi-discipline multimodal understanding benchmark. *arXiv preprint arXiv:2409.02813*, 2024. URL https://arxiv.org/abs/2409.02813.

Oliver Zahn and Simran Chana. Facts as first class objects: Knowledge objects for persistent llm memory. *arXiv preprint arXiv:2603.17781*, 2026. URL https://arxiv.org/abs/2603.17781.

Weiqi Zhai et al. Hle-verified: A systematic verification and structured revision of humanity's last exam. *arXiv preprint arXiv:2602.13964*, 2026. URL https://arxiv.org/abs/2602.13964.

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

Haozhen Zhang, Haodong Yue, Tao Feng, Quanyu Long, Jianzhu Bao, Bowen Jin, Weizhi Zhang, Xiao Li, Jiaxuan You, Chengwei Qin, and Wenya Wang. Learning query-aware budget-tier routing for runtime agent memory. *arXiv preprint arXiv:2602.06025*, 2026b. URL https://arxiv.org/abs/2602.06025.

Linfeng Zhang, Siheng Chen, Yuzhu Cai, et al. Bohrium + scimaster: Building the infrastructure and ecosystem for agentic science at scale. *arXiv preprint arXiv:2512.20469*, 2025a. URL https://arxiv.org/abs/2512.20469.

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. *arXiv preprint arXiv:2510.04618*, 2025b. URL https://arxiv.org/abs/2510.04618.

Ruijia Zhang, Xinyan Zhao, Ruixiang Wang, Sigen Chen, Guibin Zhang, An Zhang, Kun Wang, and Qingsong Wen. Safesieve: From heuristics to experience in progressive pruning for llm-based multi-agent communication. *arXiv preprint arXiv:2508.11733*, 2025c. URL https://arxiv.org/abs/2508.11733.

Shengtao Zhang, Jiaqian Wang, Ruiwen Zhou, Junwei Liao, Yuchen Feng, Weinan Zhang, Ying Wen, Zhiyu Li, Feiyu Xiong, Yutao Qi, Bo Tang, and Muning Wen. Memrl: Self-evolving agents via runtime reinforcement learning on episodic memory. *arXiv preprint arXiv:2601.03192*, 2026c. URL https://arxiv.org/abs/2601.03192v1.

Junhao Zheng, Chengming Shi, Xidi Cai, Qiuke Li, Duzhen Zhang, Chenxing Li, Dong Yu, and Qianli Ma. Lifelong learning of large language model based agents: A roadmap. *arXiv preprint arXiv:2501.07278*, 2025a. URL https://arxiv.org/abs/2501.07278.

Yingwei Zheng, Cong Li, Shaohua Li, Yuqun Zhang, and Zhendong Su. Agentic harness for real-world compilers. *arXiv preprint arXiv:2603.20075*, 2026. URL https://arxiv.org/abs/2603.20075.

Yusheng Zheng, Yanpeng Hu, Tong Yu, and Andi Quinn. Agentsight: System-level observability for ai agents using ebpf. *arXiv preprint arXiv:2508.02736*, 2025b. URL https://arxiv.org/abs/2508.02736.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *ICML*, 2024. URL https://proceedings.mlr.press/v235/zhou24r.html.

Chenyu Zhou, Huacan Chai, Wenteng Chen, Zihan Guo, Rong Shan, Yuanyi Song, Tianyi Xu, Yingxuan Yang, Aofan Yu, Weiming Zhang, Congming Zheng, Jiachen Zhu, Zeyu Zheng, Zhuosheng Zhang, Xingyu Lou, Changwang Zhang, Zhihui Fu, Jun Wang, Weiwen Liu, Jianghao Lin, and Weinan Zhang. Externalization in llm agents: A unified review of memory, skills, protocols and harness engineering. *arXiv preprint arXiv:2604.08224*, 2026a. URL https://arxiv.org/abs/2604.08224.

Huichi Zhou, Yihang Chen, Siyuan Guo, Xue Yan, Kin Hei Lee, Zihan Wang, Ka Yiu Lee, Guchun Zhang, Kun Shao, Linyi Yang, and Jun Wang. Memento: Fine-tuning llm agents without fine-tuning llms. *arXiv preprint arXiv:2508.16153*, 2025a. URL https://arxiv.org/abs/2508.16153.

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 *ICLR*, 2026b. URL https://openreview.net/forum?id=41xrZ3uGuI.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023. URL https://arxiv.org/abs/2307.13854.

Zhanke Zhou, Chentao Cao, Xiao Feng, Xuan Li, Zongze Li, Xiangyu Lu, Jiangchao Yao, Weikai Huang, Tian Cheng, Jianghangfan Zhang, Tangyu Jiang, Linrui Xu, Yiming Zheng, Brando Miranda, Tongliang Liu, Sanmi Koyejo, Masashi Sugiyama, and Bo Han. AlphaApollo: A system for deep agentic reasoning. *arXiv preprint arXiv:2510.06261*, 2025b. URL https://arxiv.org/abs/2510.06261.

Kunlun Zhu, Hongyi Du, Zhaochen Hong, Xiaocheng Yang, Shuyi Guo, Zhe Wang, Zhenhailong Wang, Cheng Qian, Xiangru Tang, Heng Ji, and Jiaxuan You. Multiagentbench: Evaluating the collaboration and competition of llm agents. In *ACL*, 2025a. URL https://aclanthology.org/2025.acl-long.421/.

Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, Xiaoteng Ma, Xiaodong Yu, Gowtham Ramesh, Jialian Wu, Zicheng Liu, Pan Lu, James Zou, and Jiaxuan You. Where llm agents fail and how they can learn from failures. *arXiv preprint arXiv:2509.25370*, 2025b. URL https://arxiv.org/abs/2509.25370.

Qiming Zhu, Shunian Chen, Rui Yu, Zhehao Wu, and Benyou Wang. From lossy to verified: A provenance-aware tiered memory for agents. *arXiv preprint arXiv:2602.17913*, 2026. URL https://arxiv.org/abs/2602.17913.

APPENDIX

A System Design and Implementation Details **22**

A.1 Overview . . . 22
A.2 Benchmark Module . . . 24
A.3 Agent Module . . . 26
A.4 Environment Module . . . 29
A.5 Scorer Module . . . 30
A.6 Runner and Execution Records . . . 32
A.7 Run Configuration and Inspection . . . 34

B Experimental Setup and Reproducibility **35**

B.1 Evaluation Setup . . . 35
B.2 Cost Measurement and Observed Task Sets . . . 36
B.3 Reproduction Resources and Configuration Scope . . . 36

C Macro Study: Analysis Methods and Supplementary Results **37**

C.1 Benchmark Performance and Task Coverage . . . 37
C.2 Error Patterns in Failed Attempts . . . 38
C.3 Functional-Step Annotation and Aggregation . . . 42
C.4 Token-Entropy Patterns . . . 43
C.5 Token Use and Wall-Time Sensitivity . . . 45

D Component Intervention Protocols **46**

D.1 Single-Component Configuration Matrix . . . 46
D.2 Tool Registry Removal . . . 46
D.3 Skill Loading . . . 46
D.4 Memory Mechanisms and Persistence . . . 46
D.5 Costs Across Full Harness Configurations . . . 47

E Related Work **48**

E.1 Reasoning Foundation Models . . . 48
E.2 Agentic Reasoning Systems and Open Agents . . . 48
E.3 Reasoning and Interactive Evaluation Benchmarks . . . 51
E.4 Evaluation Protocols, Reliability, and Human–Agent Interaction . . . 52
E.5 From Model-Only to Model–Harness Evaluation . . . 52

A System Design and Implementation Details

We present **AlphaDiana**, a modular framework for *system-level evaluation* of LLM-based reasoning systems. AlphaDiana is designed for settings in which performance depends not only on the base models, but also on the surrounding runtime stack, including the agent harnesses, tool interfaces, execution environments, scoring protocols, and run configurations. The framework factorizes an evaluation into independent modules with explicit interfaces, enabling controlled comparison across direct models and harnessed-agent systems.

Relation to the main-text architecture. Sec. 2 describes six runtime responsibilities; this appendix documents the reusable interfaces that implement them. Configuration selects the benchmark, agent, environment, scorer, and run policy. The runner coordinates task execution and result storage; environment sessions support isolation and controlled reuse; agent adapters preserve native harness execution and access the configured inference endpoint. Outcome scorers and offline trajectory analyzers consume the resulting records. The interfaces below complement that runtime view. Experimental settings are specified in Appendix B, analysis procedures in Appendix C, and intervention policies in Appendix D.

A.1 Overview

Experiment tuple. An experiment executed by AlphaDiana is formalized as

$$\mathcal{R} = \mathcal{U}(\mathcal{B}, f_\theta, \mathcal{H}, \mathcal{E}, \mathcal{S}, \mathcal{C}),$$

where $\mathcal{R}$ is the result records of the experiment, $\mathcal{U}$ is the runner that orchestrates the experiment, $\mathcal{B}$ is the benchmark, f_θ is the base model, $\mathcal{H}$ is the harness, $\mathcal{E}$ is an optional execution environment, $\mathcal{S}$ is the scorer, and $\mathcal{C}$ is the run configuration. The agent system is denoted by

$$\mathcal{A}_{\theta,\mathcal{H}} = \mathcal{H} \odot f_\theta,$$

where $\odot$ indicates that the harness wraps the base model by providing prompts, tools, state management, control logic, and answer extraction. The direct-model baseline is the special case in which $\mathcal{H}$ reduces to minimal task formatting without an external tool loop.

Modular decomposition. AlphaDiana separates evaluation into four user-specified components (benchmark, agent, environment, scorer) and one orchestration layer (runner). Specifically, the benchmark poses tasks, the agent solves them, the environment executes tool-call requests when tool use is required, the scorer judges the resulting prediction, and the runner coordinates the full protocol. After execution, trajectory-analysis utilities derive process measurements from the saved result records. The runner executes the pipeline

$$\texttt{load task} \longrightarrow \texttt{solve} \longrightarrow \texttt{score} \longrightarrow \texttt{save results}.$$

This separation is central to the framework: benchmark-specific logic is not embedded in the harness, scorer-specific normalization is not delegated to the agent, and environment-specific execution details are isolated behind a sandbox interface. As a result, the same benchmark and scorer can be reused across multiple model-harness systems, making observed differences more attributable to the evaluated system rather than to hidden evaluation-side implementation choices.

Interaction loop. For a task instance $(x, y) \in \mathcal{B}$, the harness constructs the initial model context

$$p_1 = \mathrm{Prompt}_{\mathcal{H}}(x),$$

and initializes any runtime state

$$m_1 = \mathrm{InitState}_{\mathcal{H},\mathcal{E}}(x, \mathcal{C}).$$

At turn t, the model produces

$$o_t = f_\theta(p_t).$$

The harness first classifies o_t under its native output-precedence rules. If it is classified as a terminal answer, the harness extracts the model-predicted answer $\hat{y}$ and terminates.

Otherwise, the harness attempts to parse an action candidate

$$(\tilde{a}_t, q_t) = \mathrm{TryParseAction}_{\mathcal{H}}(o_t).$$

Notation	Meaning
$\mathcal{B}$	Benchmark, represented as a finite collection of task instances.
(x, y)	Task input x and reference object y.
f_θ	Base language model with parameters θ.
$\mathcal{H}$	Harness that prompts, controls, and operationalizes f_θ.
$\mathcal{A}_{\theta,\mathcal{H}}$	Agent system obtained by pairing f_θ with $\mathcal{H}$.
$\mathcal{E}$	Optional environment backend or sandbox for tool execution.
$\mathcal{S}$	Post-execution evaluation and analysis, comprising outcome scoring and trajectory-level process analysis.
$\mathcal{C}$	Run configuration specifying budgets, policies, and logging options.
$\mathcal{U}$	Runner that orchestrates the experiment of evaluation.
$\mathcal{R}$	Result records of the experiment.
$p_t, o_t, k_t, \tilde{a}_t,$ a_t, q_t, r_t, m_t	Model-facing context, model output, native output kind, action candidate, admissible action, parse/validation status, environment or harness feedback, and runtime state at turn t.
$\hat{y}$	Final predicted answer.
τ	Complete interaction trajectory.
s	Task-level score or verdict.

Table 1: Core notation used in the AlphaDiana evaluation pipeline.

Here $\tilde{a}_t$ is the candidate (or $\varnothing$ when none is parseable), while q_t retains the native parse or validation status. Only if parsing succeeds and the requested operation is exposed and permitted does the harness accept $a_t = \tilde{a}_t$, dispatch it, and receive environment feedback

$$r_t = \mathcal{E}(a_t).$$

If parsing or validation fails, $a_t = \varnothing$ and the native runtime instead returns invalid-action feedback or records a terminal error without dispatching to $\mathcal{E}$. Whenever feedback is returned, the harness updates the runtime state and next model-facing context:

$$m_{t+1} = \mathrm{StateUpdate}_{\mathcal{H}}(m_t, o_t, r_t), \qquad p_{t+1} = \mathrm{PromptUpdate}_{\mathcal{H}}(p_t, o_t, r_t, m_{t+1}).$$

Thus, one admissible-action interaction step can be summarized as

$$p_t \xrightarrow{f_\theta} o_t \xrightarrow{\mathcal{H}} a_t \xrightarrow{\mathcal{E}} r_t \xrightarrow{\mathrm{PromptUpdate}_{\mathcal{H}}} p_{t+1}.$$

Trajectory and scoring. The full trajectory is

$$\tau = \big((p_1, o_1, a_1, r_1), \ldots, (p_T, o_T, a_T, r_T), \hat{y}\big),$$

where T is the number of turns before termination and $a_t = \varnothing$ on terminal or non-action turns. Termination occurs when the model emits a final answer, a budget in $\mathcal{C}$ is exhausted, or the harness or environment reaches a terminal error state. After termination, the scorer evaluates the prediction independently of the generation process:

$$s = \mathcal{S}(x, y, \hat{y}, \tau).$$

The runner then stores the prediction, score, trajectory, and metadata for later analysis.

Full procedure. Algorithm 1 summarizes the full evaluation procedure.

Algorithm 1 The Evaluation Pipeline of AlphaDiana

Require: Benchmark $\mathcal{B}$, base model f_θ, harness $\mathcal{H}$, environment $\mathcal{E}$, scorer $\mathcal{S}$, configuration $\mathcal{C}$
Ensure: Result records $\mathcal{R}$ containing predictions, scores, trajectories, and metadata

Instantiate the agent system $\mathcal{A}_{\theta,\mathcal{H}} \leftarrow \mathcal{H} \odot f_\theta$
Initialize result store $\mathcal{R} \leftarrow \emptyset$
for each task instance $(x, y) \sim \mathcal{B}$ **do**
  Construct initial prompt $p_1 \leftarrow \mathrm{Prompt}_{\mathcal{H}}(x)$
  Initialize harness state $m_1 \leftarrow \mathrm{InitState}_{\mathcal{H},\mathcal{E}}(x, \mathcal{C})$
  Initialize trajectory $\tau \leftarrow \emptyset$, turn index $t \leftarrow 1$, final prediction $\hat{y} \leftarrow \varnothing$
  while not terminated under $\mathcal{C}$ **do**
    Generate model output $o_t \leftarrow f_\theta(p_t)$
    Classify output kind $k_t \leftarrow \mathrm{OutputKind}_{\mathcal{H}}(o_t)$
    if k_t is a terminal answer **then**
      Extract prediction $\hat{y} \leftarrow \mathrm{ExtractAnswer}_{\mathcal{H}}(o_t)$
      Append terminal record $(p_t, o_t, \varnothing, \varnothing)$ to τ
      break
    end if
    Attempt action parsing $(\tilde{a}_t, q_t) \leftarrow \mathrm{TryParseAction}_{\mathcal{H}}(o_t)$
    if q_t is parsed and $\tilde{a}_t$ invokes an exposed, policy-permitted operation **then**
      Accept action $a_t \leftarrow \tilde{a}_t$
      Execute action and obtain feedback $r_t \leftarrow \mathcal{E}(a_t)$
    else
      Reject candidate $a_t \leftarrow \varnothing$
      Set native error record $r_t \leftarrow \mathrm{HandleInvalidAction}_{\mathcal{H}}(o_t, \tilde{a}_t, q_t)$
    end if
    Append turn record (p_t, o_t, a_t, r_t) to τ
    if r_t marks a terminal error **then**
      Record the native finish reason
      break
    end if
    Update harness state $m_{t+1} \leftarrow \mathrm{StateUpdate}_{\mathcal{H}}(m_t, o_t, r_t)$
    Update prompt/context $p_{t+1} \leftarrow \mathrm{PromptUpdate}_{\mathcal{H}}(p_t, o_t, r_t, m_{t+1})$
    Increment turn index $t \leftarrow t + 1$
  end while
  if $\hat{y} = \varnothing$ **then**
    Extract fallback prediction $\hat{y} \leftarrow \mathrm{FallbackExtract}_{\mathcal{H}}(\tau)$
  end if
  Score prediction $s \leftarrow \mathcal{S}(x, y, \hat{y}, \tau)$
  Collect metadata $z \leftarrow \mathrm{Metadata}(\mathcal{B}, f_\theta, \mathcal{H}, \mathcal{E}, \mathcal{S}, \mathcal{C}, \tau)$
  Save result $\mathcal{R} \leftarrow \mathcal{R} \cup \{(x, y, \hat{y}, s, \tau, z)\}$
end for

return $\mathcal{R}$

A.2 Benchmark Module

Role. The benchmark module loads a dataset and materializes it into standardized task instances. Formally, a benchmark is a finite collection

$$\mathcal{B} = \{(x_i, y_i)\}_{i=1}^{N},$$

where x_i is the task input and y_i is the reference object used for scoring. The reference object may be a string, numeric answer, symbolic expression, unit-test specification, verifier target, rubric, set of acceptable answers, or any other task-specific supervision object. AlphaDiana treats y_i abstractly: the benchmark provides the reference, while the scorer determines how it should be interpreted.

Task record. In implementation, each task record contains a unique task identifier, input x, reference field y, metadata, and optional attachments. Metadata may include difficulty, category, source, split, or expected output type. Attachments may include files, images, repositories, PDFs, or other artifacts required by multimodal or environment-based tasks. These fields may be utilized by the runner, agent, environment, or scorer, but they do not alter the core agent-facing interface.

Module boundary. The benchmark specifies *what* problem is to be solved, not *how* it should be solved. It therefore does not contain harness-specific prompts, tool policies, memory rules, recovery

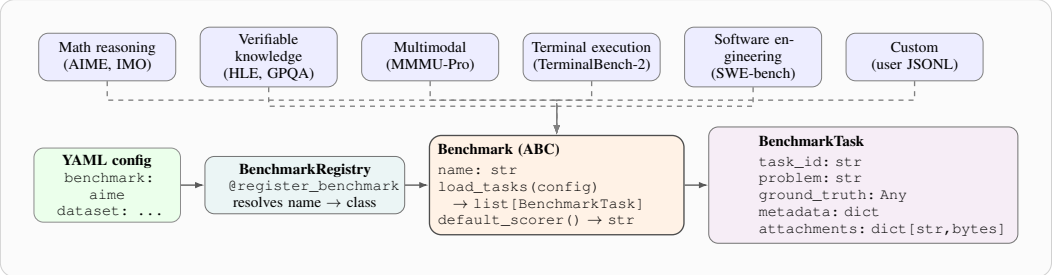


Fig. 17: **Benchmark Module.** Concrete benchmark implementations (left) subclass the abstract `Benchmark` interface, which exposes `load_tasks()` to convert a raw dataset into a sequence of standardized `BenchmarkTask` tuples (right). The registry resolves a benchmark name from the YAML configuration to the corresponding implementation at runtime.

strategies, or scoring logic. Prompt construction belongs to the harness, execution belongs to the environment, and correctness checking belongs to the scorer. This separation prevents benchmark definitions from implicitly favoring a particular harness design.

Interface. The abstract `Benchmark` interface exposes one required method:

`load_tasks(config)` → `list[BenchmarkTask]`.

This method converts a raw dataset, file collection, benchmark API, or generated task set into the standardized task sequence utilized by the runner. The configuration may specify split names, subset sizes, seeds, task filters, attachment paths, or benchmark-specific preprocessing options. Adding a new benchmark requires implementing this method and registering the benchmark under a configuration-visible name; no agent, environment, scorer, or runner code needs to change.

Listing 1: Benchmark interface (`base.py`, verbatim) and a minimal custom implementation. The `@register_benchmark` decorator registers the class under a string key that matches the `benchmark:` field in the YAML configuration.

```python
# --- base.py (verbatim) ----------------------------------
@dataclass
class BenchmarkTask:
    task_id: str
    problem: str
    ground_truth: Any
    metadata: dict = field(default_factory=dict)
    attachments: dict[str, bytes] = field(default_factory=dict)

class Benchmark(ABC):
    name: str = ""

    @abstractmethod
    def load_tasks(self, config: dict) -> list[BenchmarkTask]: ...

    def default_scorer(self) -> str:
        return "exact_match"

# --- custom implementation (minimal example) --------------
@register_benchmark("my_benchmark")
class MyBenchmark(Benchmark):
    name = "my_benchmark"

    def load_tasks(self, config: dict) -> list[BenchmarkTask]:
        raw = load_my_dataset(config["dataset"])
        return [
            BenchmarkTask(
                task_id=item["id"],
                problem=item["question"],
                ground_truth=item["answer"],
```

```
            )
            for item in raw
        ]
```

A.3 AGENT MODULE

Role. The agent module provides a uniform interface over heterogeneous reasoning systems. It hides implementation differences among direct model calls, lightweight scaffolds, and full agent runtimes behind a common `solve` interface. Given a task instance, the agent returns an `AgentResponse` containing the final prediction, interaction trajectory, termination reason, and execution metadata.

Model-harness system. The evaluated system is the model-harness pair $\mathcal{A}_{\theta,\mathcal{H}}$. The harness is not merely a prompt template: it determines the initial context, admissible actions, state initialization, parsing rules, error handling, tool exposure, and update logic. It also mediates the distinction between information placed in the model context and information maintained in the external runtime. For example, a prompt may describe available tools, while the actual tool registry, workspace state, and execution constraints are managed outside the model.

Direct and harnessed modes. AlphaDiana supports both direct and harnessed modes through the same external interface. In direct mode, the system performs a single model call with minimal task formatting, extracts an answer, and returns a degenerate one-turn trajectory. In harnessed mode, the system executes the multi-turn model-harness-environment loop defined in Sec. A.1. The harness may expose tools, manage persistent state, recover from invalid actions, incorporate environment feedback, and decide how observations enter later model contexts.

AgentResponse. The `AgentResponse` object records both the final prediction and the provenance of the interaction that produced it. At minimum, it contains the predicted answer $\hat{y}$, trajectory τ, and finish reason. It may also include token usage, wall-clock time, number of turns, number of tool calls, raw model responses, exposed tools, error states, workspace artifacts, and environment snapshots. These fields make it possible to analyze not only whether the final answer was correct, but also how it was produced.

Inference endpoint and recording. The native harness queries the configured local or remote model service independently of the execution backend. Request and response recording, including token probabilities when available, supplies the evidence for the offline analysis in Appendix C. A missing probability or usage field limits the corresponding measurement; it does not remove the attempt from outcome accounting.

Module boundary. The agent is separated from both the benchmark and the scorer. The benchmark provides the task input and reference object, but does not prescribe the agent's reasoning strategy. The scorer evaluates the prediction after generation, but does not participate in prompting, tool use, or recovery. This boundary is necessary for controlled cross-harness comparison: agents may differ in how they solve tasks, but they are evaluated against the same task and scoring interface.

Interface. Adding a new agent requires implementing:

`setup(config)`

for initializing the model client, harness configuration, prompt templates, tool registry, and runtime state; and

`solve(task, environment)` $\rightarrow$ `AgentResponse`

for executing a task and returning the prediction, trajectory, and metadata. Once registered, the agent can be selected directly from the YAML experiment specification.

Listing 2: Agent interface (`base.py`, verbatim) and a minimal custom implementation.

```python
# --- base.py (verbatim) ----------------------------------
@dataclass
class AgentResponse:
    answer: Any                          # final prediction
    trajectory: list[dict] = ...         # action steps
    reasoning_trajectory: list[dict] = ... # reasoning trace
    token_usage: dict = ...              # cost metrics
```

```
    wall_time_sec: float = 0.0
    artifact_manifest: dict = ...        # workspace artifacts
    system_prompt: str = ""              # harness observability
    # ... (further provenance fields)

class Agent(ABC):
    @abstractmethod
    def setup(self, config: dict) -> None: ...

    @abstractmethod
    def solve(self, task: BenchmarkTask,
              sandbox: Optional[Any] = None) -> AgentResponse: ...

# --- custom implementation (minimal example) --------------
@register_agent("my_agent")
class MyAgent(Agent):
    def setup(self, config: dict) -> None:
        self.model = config["model"]

    def solve(self, task, sandbox=None) -> AgentResponse:
        answer = call_my_model(self.model, task.problem)
        return AgentResponse(answer=answer)
```

A.3.1 Matched Harness Trace

The common loop aligns corresponding execution stages while preserving each harness's native behavior. Tab. 2 follows one matched example, Qwen3.5-27B on HLE item `hle_1792`. All three harnesses use local computation, obtain $\min b(t) \approx 1.0346$, and select answer E, but they reach that answer through different parsing, validation, feedback, and context-update paths.

Table 2: **Matched native execution trace for one model and task.** The columns align the three harnesses to the common loop without replacing their native events or state representations.

Harness	Model-facing context and proposal	Validation and feedback	Update and outcome
OpenClaw	Gateway conversation produces a native local-computation request.	The controller dispatches the request; an `odeint` error returns as structured feedback.	The controller preserves the feedback, recovers with a corrected computation, and the model selects E.
ZeroClaw	CLI history produces a native action proposal.	Shell policy rejects one proposal before later file and shell actions are accepted.	The rejection and accepted-action results enter the expanded history, after which the model selects E.
OpenCode	Session messages contain a typed `bash` proposal.	The CLI permission path accepts the tool request and records its typed result.	The completed result remains in the rendered session context, after which the model selects E.

The example separates the shared evaluation contract from the system under study. AlphaDiana fixes the task, model, scorer, expected work, and saved fields, while each harness retains its native action grammar, approval path, error handling, and context representation. The standardized trajectory makes these stages comparable without requiring the three harnesses to execute through the same internal protocol.

A.3.2 Measured Extension Footprints

The module boundary also localizes the work required to add a domain or harness. Tab. 3 reports measured implementation footprints for three completed extensions. The counts describe the code added for these integrations, not universal person-hour estimates; the required work depends on whether the extension can reuse an existing environment and verifier.

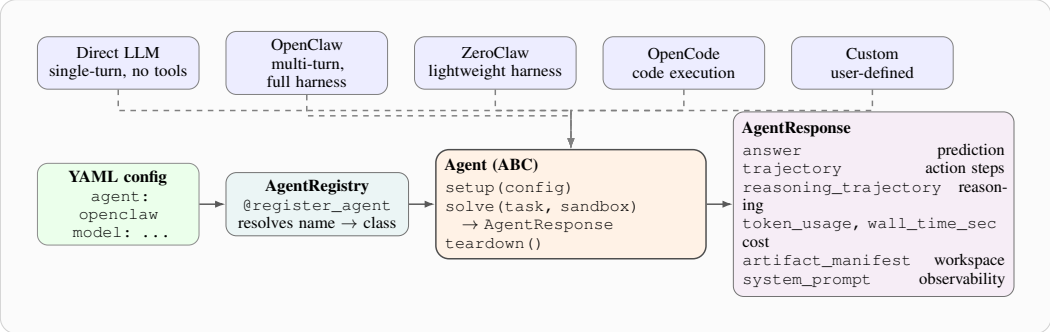


Fig. 18: **Agent Module.** Concrete agent implementations (top) subclass the abstract `Agent` interface, which exposes `solve(task, sandbox)` → `AgentResponse`. The registry resolves an agent name from the YAML configuration at runtime. `AgentResponse` captures the full execution provenance: prediction, action trajectory, reasoning trace, cost metrics, workspace artifacts, and harness observability fields.

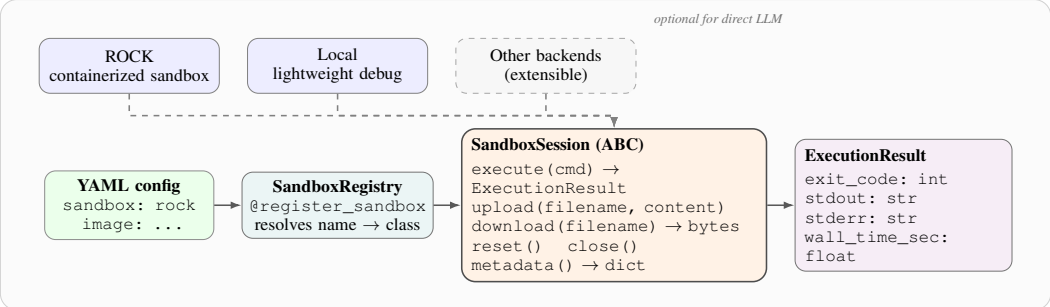


Fig. 19: **Environment Module.** Backend implementations (top) subclass the abstract `SandboxSession` interface. The sandbox is optional: direct LLM evaluation requires no execution backend; harnessed agents (*e.g.*, OpenClaw) use a ROCK-based containerized sandbox. The interface contract — `execute`, `upload`/`download`, `reset`, `close`, `metadata` — is the portability boundary that makes backends interchangeable.

Table 3: **Measured implementation footprints for three extension scenarios.** Shared runner and result-storage code is reused in every case.

Extension scenario	Components changed	Measured footprint
Add MMMU-Pro as a non-interactive domain	Benchmark	202 lines
Extend OpenClaw to Terminal-Bench 2	Benchmark, environment, and OpenClaw integration	∼2,150 shared + ∼850 harness-specific lines
Add Pi to the supported AIME 2024 domain	Agent	∼520 lines + ∼45-line configuration

A compatible non-interactive domain may require only a benchmark adapter, as in MMMU-Pro. An interactive domain additionally requires task-environment and verifier support, as in Terminal-Bench 2. A new harness on a supported domain primarily requires an agent adapter, as in Pi on AIME 2024; an additional environment integration is needed only when the harness cannot reuse an existing backend. In all three cases, the Runner and standardized result format remain unchanged.

A.4 Environment Module

Role. The environment module provides the external execution substrate for agents that need to run code, access files, invoke tools, or interact with task-specific artifacts. It is optional for direct-model evaluation but becomes a first-class component in harnessed settings because it determines where actions are executed, what resources are available, what limits apply, and what isolation boundary separates agent behavior from the evaluation host.

Module boundary. The runner creates and manages the environment session, then passes it to the agent during task execution. The benchmark and scorer do not interact with the environment directly. This preserves the division of responsibilities: the benchmark specifies the task, the agent selects actions, the environment executes admissible actions, and the scorer evaluates the final prediction.

Session interface. An environment backend is exposed through a `SandboxSession` interface with the following core operations:

$$\texttt{execute(command)} \rightarrow \texttt{ExecutionResult},$$
$$\texttt{upload(path, content)}, \quad \texttt{download(path)},$$
$$\texttt{reset()}, \quad \texttt{close()}, \quad \texttt{metadata()} \rightarrow \texttt{dict}.$$

These operations support command execution, file transfer, task-level cleanup, session split, and provenance reporting. Any backend implementing this interface is interchangeable in AlphaDiana.

State reset and reproducibility. The `reset()` operation is important for reproducibility. Agentic tasks often modify the workspace by creating files, installing packages, writing logs, editing repositories, or caching intermediate outputs. Without explicit reset semantics, later tasks may inherit hidden state from earlier tasks, making results order-dependent. AlphaDiana, therefore, treats reset behavior as part of the environment contract. A backend may implement reset by restoring a filesystem snapshot, clearing temporary directories, restarting services, or recreating task-local state.

Session allocation and controlled reuse. As described in Sec. 2, parallel attempts acquire separate sessions when the backend supports pooling, and sessions are reset before reassignment. Under strict isolation, failed sandbox setup or incomplete pool preparation stops the run. Sequential attempts may reuse a session, while the adapter separately controls working files and native memory. Sharing within a task reinitializes the agent and replaces the shared session before the next task. Memory-sharing modes require sequential execution and disable automatic retries because a failed attempt may already have changed memory. The experimental persistence scopes are specified in Appendix D.4.

ROCK backend. The default implementation uses a ROCK-based containerized backend. ROCK isolates agent actions from the evaluation host, confines command execution and file edits to a sandbox boundary, records resource profiles and image provenance, and supports retrieval of workspace artifacts after completion. This standardizes the execution substrate across runs and reduces variation due to local machine state, dependency drift, or uncontrolled filesystem differences. These properties are especially important for evaluating agents on tasks involving code execution, terminal interaction, repository modification, or artifact generation.

Listing 3: Sandbox interface (`base.py`, verbatim) and a minimal custom backend.

```python
# --- base.py (verbatim) ----------------------------------
@dataclass
class ExecutionResult:
    exit_code: int
    stdout: str
    stderr: str
    wall_time_sec: float

class SandboxSession(ABC):
    @abstractmethod
    def execute(self, command: str) -> ExecutionResult: ...
    @abstractmethod
    def upload(self, filename: str, content: bytes) -> None: ...
    @abstractmethod
    def download(self, filename: str) -> bytes: ...
    @abstractmethod
```

```
    def close(self) -> None: ...
    def reset(self) -> None: ...          # clear state between tasks
    def metadata(self) -> dict: ...      # backend provenance

# --- custom backend (minimal example) --------------------
@register_sandbox("my_sandbox")
class MySandboxSession(SandboxSession):
    def execute(self, command: str) -> ExecutionResult:
        result = subprocess.run(command, shell=True, capture_output=
            True)
        return ExecutionResult(
            exit_code=result.returncode,
            stdout=result.stdout.decode(),
            stderr=result.stderr.decode(),
            wall_time_sec=0.0,
        )
    def upload(self, filename, content): ...
    def download(self, filename): ...
    def close(self): ...
```

A.5 Scorer Module

Role. At the conceptual level, $\mathcal{S}$ denotes two post-execution layers. The *outcome-scoring layer* determines whether the final prediction solves the task and returns the task-level judgment s. The *process-analysis layer* reads the saved trajectory τ and run metadata from $\mathcal{R}$ to measure action composition, token entropy, token and time use, completion, and failures. Both layers are external to generation: neither controls prompts, tool use, environment interaction, recovery behavior, or answer generation.

In the implementation, the abstract `Scorer` interface realizes the outcome-scoring layer. Given the task input x, reference object y, and prediction $\hat{y}$, it applies a shared verification rule and returns a `ScoreResult`. Trajectory analyzers remain separate utilities and run over persisted records after execution. This separation keeps correctness independent of harness-specific self-judgment while allowing outcome and process measurements to be reported together.

ScoreResult. The output is a `ScoreResult`, which contains a binary `correct` label, a scalar `score`, the expected reference value, the predicted value, a rationale, and metadata. The binary label supports accuracy-style metrics, while the scalar score supports partial credit or continuous-valued grading. Metadata may store normalization details, verifier outputs, judge responses, tolerance settings, error messages, or failure labels.

Built-in scoring strategies. AlphaDiana supports three built-in scoring strategies. Exact-match scoring compares normalized strings or categorical labels:

$$\texttt{correct} = \mathbf{1}[\mathrm{norm}(\hat{y}) = \mathrm{norm}(y)]\,.$$

Math-aware scoring first extracts the final answer and then applies symbolic, numeric, interval, unit, or equivalence checks:

$$\texttt{correct} = \mathbf{1}\big[\mathrm{verify}\big(\mathrm{extract}(\hat{y}), y\big)\big]\,.$$

LLM-as-judge scoring delegates the judgment to an independent judge model or judge pipeline,

$$\texttt{ScoreResult} = J_\phi(x, y, \hat{y}),$$

using a fixed rubric, prompt, model, and decoding configuration recorded in the scorer metadata. This strategy is used when deterministic matching is too brittle, when multiple acceptable responses exist, or when partial-credit rubrics are required.

Correctness versus system-level measurement. The outcome scorer determines task-level correctness, while the runner records the trajectory and system-level measurements required by the process-analysis layer, including latency, token usage, number of turns, tool-call counts, timeout rate, crash rate, and completion rate. Post-hoc analyzers additionally compute action and entropy statistics from these records. Keeping the two layers distinct in the implementation allows systems

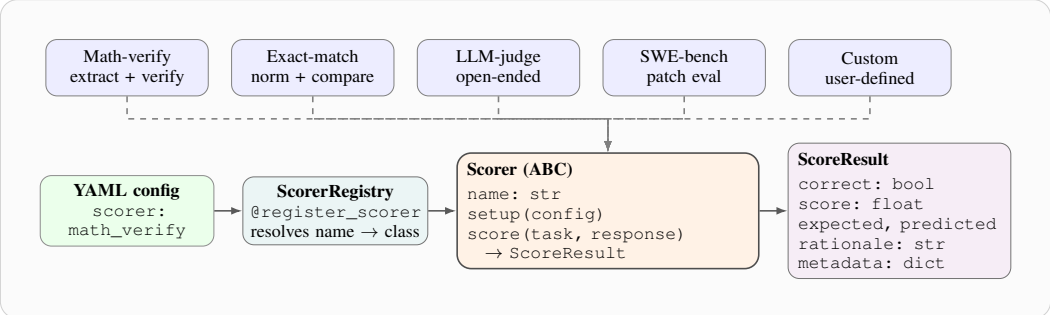


Fig. 20: **Outcome-scoring layer of $\mathcal{S}$.** Scorer implementations subclass the abstract `Scorer` interface, which maps a task and agent response to a `ScoreResult`. This interface implements task-level outcome judgment; the complementary process-analysis layer reads persisted trajectories and metadata from $\mathcal{R}$ after execution.

to be compared along multiple axes without conflating accuracy with cost, reliability, or trajectory behavior.

Interface. Adding a new scorer requires implementing:

$$\texttt{score(task, response)} \rightarrow \texttt{ScoreResult}.$$

The `task` argument provides the input, reference object, identifiers, metadata, and attachments; the `response` argument provides the prediction and agent-side metadata together with the stored trajectory when available. Outcome scorers consume the fields required by their verification rule, while process analyzers operate on the resulting records. Once registered, a scorer can be selected from the YAML experiment specification without modifying the benchmark, agent, environment, or runner.

Listing 4: Scorer interface (`base.py`, verbatim) and a minimal custom implementation.

```python
# --- base.py (verbatim) ------------------------------------
@dataclass
class ScoreResult:
    correct: bool
    score: float
    expected: Any
    predicted: Any
    rationale: str = ""
    metadata: dict = field(default_factory=dict)

class Scorer(ABC):
    @property
    @abstractmethod
    def name(self) -> str: ...

    def setup(self, config: dict) -> None: ...  # optional

    @abstractmethod
    def score(self, task: BenchmarkTask,
              response: AgentResponse) -> ScoreResult: ...

# --- custom implementation (minimal example) --------------
@register_scorer("my_scorer")
class MyScorer(Scorer):
    @property
    def name(self) -> str:
        return "my_scorer"

    def score(self, task, response) -> ScoreResult:
        correct = response.answer == task.ground_truth
```

```
        return ScoreResult(
            correct=correct, score=float(correct),
            expected=task.ground_truth,
            predicted=response.answer,
        )
```

A.6 Runner and Execution Records

Role. The runner is the orchestration layer of AlphaDiana. It instantiates the benchmark, agent, environment, scorer, and configuration; manages the environment lifecycle; dispatches tasks to the agent; invokes the scorer after generation; and persists all information needed for later analysis. The runner does not solve tasks or judge correctness directly. Instead, it enforces the evaluation protocol specified by $\mathcal{C}$.

Result record. For each task attempt, the runner writes an append-only result record. Abstractly, the record for attempt j on task i is

$$\rho_{i,j} = \big(x_i,\, y_i,\, \hat{y}_{i,j},\, s_{i,j},\, \tau_{i,j},\, z_{i,j}\big),$$

where $\tau_{i,j}$ is the trajectory produced by that attempt and $z_{i,j}$ denotes metadata collected from the benchmark, agent, environment, scorer, and run configuration. In single-sample evaluation, each task has one attempt, so we write ρ_i as shorthand for $\rho_{i,1}$. In implementation, records are stored as JSONL entries containing task identity, attempt identity, prediction, score, trajectory, cost and efficiency metrics, environment provenance, harness observability fields, artifacts, raw response payloads, and termination information.

Trajectory-aware analysis. This schema supports trajectory-aware evaluation. It preserves not only whether the agent solved the task, but also how the solution was produced, what resources were consumed, which tools were used, what feedback was observed, and why the run terminated. Downstream analyses can therefore compute accuracy, inspect failure modes, compare harness behavior, analyze tool-use patterns, and audit resource consumption without re-running the experiment.

Failure preservation. The runner also preserves both system-level and model-level failures, including a model API call failing, an invalid action occurring, a harness crashing, an environment operation times out, a budget is exhausted, or a scorer raises an exception. The runner writes records with all available task, trajectory, metadata, error, and finish-reason fields. This prevents failed tasks from being silently discarded and allows completion rate, timeout rate, crash rate, and other reliability statistics to be computed from the same result store as accuracy.

Run-level aggregation. Let $\mathcal{R} = \{\rho_{i,j}\}$ denote the saved attempt-level records for a run, where $\rho_{i,j}$ is the record for attempt j on task i. In single-sample evaluations, each task has one attempt, so ρ_i may be used as shorthand for $\rho_{i,1}$. Standard aggregate metrics include accuracy, mean score, completion rate, timeout rate, crash rate, mean latency, token usage, number of turns, and number of tool calls. These summaries are derived from the saved records rather than from transient in-memory state, so reports can be regenerated after the run finishes. For multi-sample evaluation, AlphaDiana also supports common sampling-based metrics such as pass-rate and average correctness over repeated attempts for the same task. These metrics are computed by grouping attempt-level records by task identifier and aggregating the corresponding predictions, scores, trajectories, and metadata. This allows AlphaDiana to support both single-sample and multi-sample evaluation regimes without changing the underlying result schema.

Retries and completion accounting. The append-only store can retain multiple records for one task and sample. Reports select the latest record for that pair; a retry does not create an additional sample. Correctness and execution status remain separate, and failures and unscored outputs are retained. Measurement-specific filters, such as requiring a recorded token count or a common task intersection, are reported in Appendix B.2.

Fine-grained provenance. Some analyses require information below the task level. For example, uncertainty estimation, calibration analysis, and token-level perplexity analysis require access to model log-probabilities when available. AlphaDiana therefore preserves raw model response payloads, including log-probability fields exposed by the model API, inside the response metadata. This enables downstream computation of decision uncertainty, confidence-correctness gaps, trajectory-level volatility, and related diagnostics.

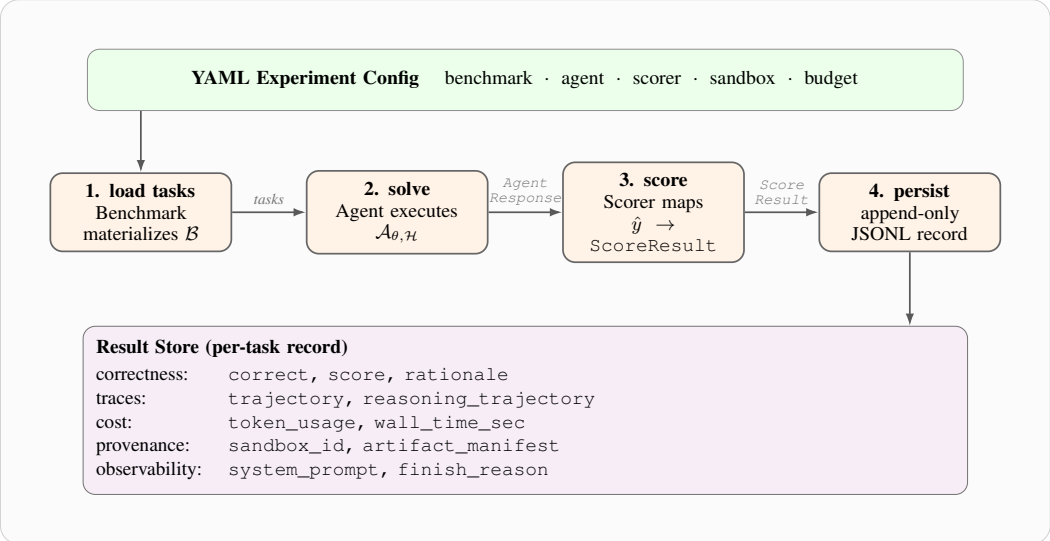


Fig. 21: **Runner.** The runner instantiates all components from a YAML configuration and executes the four-stage pipeline: load tasks → solve → score → persist. Each per-task result record captures correctness, execution traces, cost metrics, sandbox provenance, and harness observability fields in a single append-only JSONL entry.

Interface boundary. Overall, the runner coordinates modules without collapsing their responsibilities. It does not embed benchmark-specific prompting, agent-specific scoring, or scorer-specific execution logic. This boundary makes AlphaDiana experiments reproducible and comparable: different model-harness systems can share the benchmark, model, scorer, and resource policy, while any harness-coupled environment backend is recorded explicitly and all trajectories and outcomes use a common schema.

Listing 5: Minimal runner loop showing the four-stage pipeline. The actual `Runner` class handles concurrency, error recovery, and sandbox lifecycle; this excerpt shows the core per-task logic.

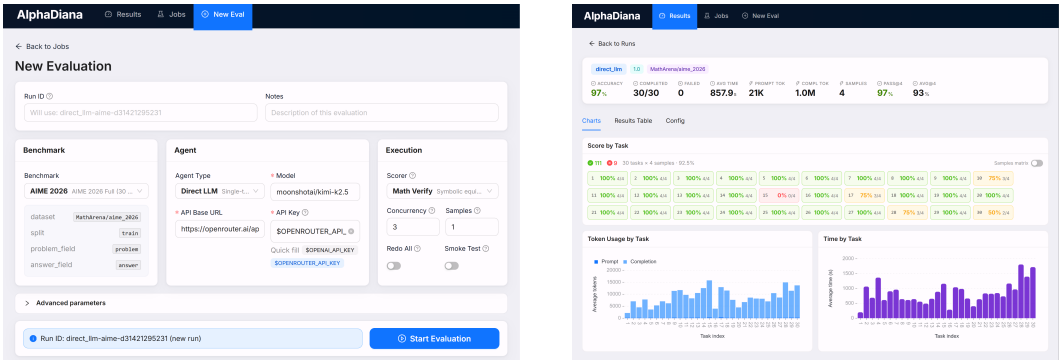
```python
def run(config: ExperimentConfig, output_dir: str) -> None:
    # instantiate components from config
    benchmark = BenchmarkRegistry.get(config.benchmark_name)()
    agent     = AgentRegistry.get(config.agent_name)()
    scorer    = ScorerRegistry.get(config.scorer_name)()
    sandbox   = SandboxRegistry.get(config.sandbox_name)() \
                if config.sandbox_name else None

    benchmark.setup(config.benchmark_config)
    agent.setup(config.agent_config)
    scorer.setup(config.scorer_config)
    if sandbox: sandbox.setup(config.sandbox_config)

    store = ResultStore(output_dir, run_id=config.run_id)

    # four-stage pipeline
    for task in benchmark.load_tasks(config.benchmark_config): # 1. load
        session = sandbox.create_session() if sandbox else None
        response = agent.solve(task, session)                  # 2. solve
        result   = scorer.score(task, response)                # 3. score
        store.append(task, response, result)                   # 4. persist
        if session: session.close()
```

A.7 Run Configuration and Inspection



(a) Create a new evaluation

(b) Obtain the evaluation results

Fig. 22: **AlphaDiana dashboard demo. (a)** The run-creation interface instantiates an evaluation from explicit configuration fields. In this example, the benchmark is AIME 2026, the agent is Direct LLM (the *Direct* baseline), and the remaining fields set the model and scorer, which correspond to the run configuration $\mathcal{C}$. **(b)** The dashboard of results $\mathcal{R}$ visualizes an executed AIME 2026 run with token and time statistics. The summary reports $30/30$ completed tasks, zero failed runs, 97% Pass@4, and 93% Avg@4.

Fig. 22 illustrates configuration and result inspection. Its displayed run is a dashboard example; the paper's experimental settings and computing resources are documented in Appendix B. Saved task and sample identifiers link outcomes to traces and permit offline analysis without rerunning inference.

B EXPERIMENTAL SETUP AND REPRODUCIBILITY

This appendix specifies the shared experimental settings, resource limits, and measurement coverage for Secs. 3 and 4. Analysis methods and supplementary macro results are collected in Appendix C; intervention-specific configurations are collected in Appendix D.

B.1 EVALUATION SETUP

Tab. 4 lists the task inputs, outcome scorers, and reported metrics of each benchmark $\mathcal{B}$. Tab. 5 shows how each evaluated system implements the three functions of a native harness in Fig. 23: the prompt and action interface, control logic, and runtime state. During execution, each harness $\mathcal{H}$ presents the task input x to the base model f_θ with its native prompts and available tools; if the model produces an admissible action, the harness sends it to $\mathcal{E}$, adds the returned feedback to the next prompt, and continues while maintaining its session, trajectory, workspace, and any native memory. Fig. 22 shows the dashboard that instantiates a run from explicit configuration fields and inspects its saved records.

Harness defaults and interactive tasks. OpenClaw, ZeroClaw, and OpenCode use default configurations, with native memory available but no explicit instruction to use it. *Direct* uses a single model response without a tool loop. Interactive evaluations use Qwen3.5-27B and Gemma-4-31B-IT: SWE Mini contains 50 SWE-bench Verified tasks and uses SWE-Agent as the reference; Terminal-Bench 2 contains 89 tasks and uses terminus-2. Their Avg@1 scores count runtime and evaluator failures as unsuccessful attempts.

Sampling, decoding, and budgets. Macro results use one sample per problem (Avg@1), except on AIME 26, which has 30 problems and draws four samples each at temperature 0.6. Pass@4 is the fraction of tasks solved at least once; Avg@4 is the mean accuracy across all attempts. Within a model–task block, systems share the task list and a decoding policy of top-p 0.95 with a per-request generation limit of `max_tokens`$=$ 131,072. Appendix C.5 gives the wall-time and iteration caps and explains why these matched limits do not equalize realized compute.

Table 4: **Benchmark inputs, scorers, and reported metrics.** Rows follow the settings reported in Fig. 5.

Benchmark $\mathcal{B}$	Task input	Outcome scorer	Reported metric
AIME 26	Text	`numeric`	Pass@4, Avg@4
IMO-AnswerBench	Text	`imo_verify`	Avg@1
GPQA-Diamond	Text	`exact_match`	Avg@1
HLE-Verifiable	Text	`exact_match`	Avg@1
MMMU-Pro	Text + image	`exact_match`	Avg@1
SWE Mini	Issue + repository	`swe_bench`	Avg@1
Terminal-Bench 2	Instruction + container	`terminal_bench2`	Avg@1

Table 5: **Harness components and settings.** The three component columns follow Fig. 23; the last two rows are the baseline systems for SWE Mini and Terminal-Bench 2.

System and setting	Prompt and action interface	Control logic	Runtime state
Direct (Base model)	Minimal task prompt; no tools or actions	Single model response; no harness control loop	No harness state
OpenClaw (2026.3.7)	Native prompt and tools; no added skills	Prompt updates, interaction loop, recovery, stopping	Session and workspace for each attempt; memory use not elicited
OpenCode (v1.3.2)	System and task prompts; native tools; no added skills	Interaction loop, run policies, recovery, stopping	New session and workspace; memory use not elicited
ZeroClaw (v0.6.9)	Native prompt and configurable tools; no added skills	Interaction loop, run policies, turn and stopping limits	Session and workspace for each task; memory use not elicited
SWE-Agent (v1.1.0; SWE Mini baseline)	SWE Mini task prompt and file editing tools	Interaction loop, inspection, editing, patch submission	Task container and trajectory; no state shared across tasks
terminus-2 (Terminal-Bench 2 baseline)	Terminal-Bench 2 task prompt and native tool calls	Interaction loop, execution, feedback, stopping	Task container; no state shared across tasks

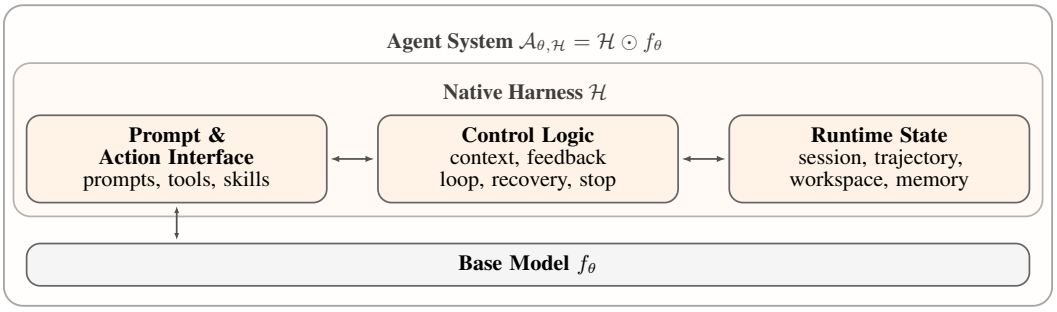


Fig. 23: **Functional anatomy of a native harness.** The interface exchanges prompts and outputs with the fixed model. The control logic advances the interaction while reading or updating runtime state. Direct uses one response, with no tools, control loop, or runtime state.

B.2 Cost Measurement and Observed Task Sets

We report recorded output tokens in thousands, excluding input tokens, and wall time in seconds. Both measures include work spent on unsuccessful attempts.

Macro comparisons. Fig. 9 uses each system's available cohort without enforcing a common task intersection. Token means include only attempts with recorded counts. Token capture is incomplete for OpenClaw and ZeroClaw on Qwen/AIME 26, all three harnesses on Qwen/GPQA-Diamond, ZeroClaw on both models/HLE-Verifiable, and OpenClaw and ZeroClaw on both models/MMMU-Pro. Gemma/MMMU-Pro/OpenClaw has 1,570 attempts, compared with 1,730 for the other systems. The two comparisons discussed in the main text have complete token coverage: 120 attempts each for Direct and OpenCode on Qwen/AIME 26, and 198 tasks each for Direct and ZeroClaw on Gemma/GPQA-Diamond. Recorded execution time includes model serving, tool execution, and scheduling. The figure summarizes observed runs rather than a standardized latency experiment. In particular, Gemma/AIME 26 source runs differ in execution date and serving endpoint. Serving configuration, concurrency, load, and timing boundaries are not fully matched across the source runs. These summaries therefore describe observed execution time, without isolating harness overhead.

Costs per solved attempt. Figs. 15 and 16 use common Gemma task sets of 198 GPQA-Diamond and 587 HLE-Verifiable tasks. Four HLE tasks with missing ZeroClaw token captures are excluded from every system. All four excluded ZeroClaw attempts were unsuccessful, so results describe the observed subset. Tokens per attempt divide total recorded output by the number of attempts. Tokens per solved attempt divide that same total, including output from unsuccessful attempts, by the number of correct answers.

Token allocation. Unsuccessful attempts include incorrect answers, invalid or unscored outputs, and operational failures. The highest-token group contains 20 of 198 GPQA attempts and 59 of 587 HLE attempts, rounding 10% upward. Its token share overlaps with the unsuccessful-attempt share, so the two shares should not be added.

B.3 Reproduction Resources and Configuration Scope

Configuration scope. A reproduced comparison must retain the model endpoint, harness version, task list, scorer, decoding settings, sample identifiers, and resource limits of its model–task block. The system interfaces and saved records are described in Appendix A. The sampling settings above and the runtime caps in Appendix C.5 define the macro operating point; Appendix D specifies the additional tool, skill, and memory settings for the micro study. Cost summaries must use the coverage and denominator conventions in Appendix B.2.

Compute resources. All AlphaDiana experiments were conducted on a local GPU cluster with 16 NVIDIA A100 GPUs, each with 80GB memory. The GPUs were used for model serving, trajectory generation, and harness-based evaluation. Since our study focuses on evaluation-time behavior rather than large-scale model training, the compute budget was mainly spent on repeated benchmark execution across models, harnesses, and sampling configurations.

C MACRO STUDY: ANALYSIS METHODS AND SUPPLEMENTARY RESULTS

This appendix follows the order of Sec. 3: benchmark performance, error patterns, functional-step patterns, token entropy, and token use and wall time. Each analysis is accompanied by its measurement procedure and supporting evidence. Evaluation settings and cost-coverage conventions appear in Appendix B; component interventions appear in Appendix D.

C.1 BENCHMARK PERFORMANCE AND TASK COVERAGE

The benchmark scores in Fig. 5 summarize outcomes. The paired comparisons below show which tasks change status, and the extraction ablation checks sensitivity to post-execution answer processing.

C.1.1 PAIRED COVERAGE AND TOKEN USE ON GPQA-DIAMOND

Task-paired accounting on Qwen3.5-27B/GPQA-Diamond separates three quantities that aggregate accuracy combines (Tab. 6): tasks added beyond *Direct*, *Direct*-correct tasks lost by each harness, and output-token use on wrong versus correct trajectories. The four-system union solves 175 of 198 tasks (88.4%), compared with 159 of 198 (80.3%) for *Direct*; each individual harness nevertheless loses more *Direct*-correct tasks than it adds from the *Direct*-wrong set.

Table 6: **Paired coverage and token use on Qwen3.5-27B/GPQA-Diamond** ($n = 198$)**.** Added/lost count tasks that change between correct and non-correct outcomes relative to *Direct*, including operational failures among non-correct outcomes; W/C is mean output tokens on wrong trajectories divided by mean output tokens on correct trajectories. Union credits a task when any of the four systems is correct.

System	Accuracy (%)	Added	Lost	W/C
Direct	80.3	–	–	2.2×
OpenClaw	66.2	8	36	11.9×
OpenCode	73.2	10	24	3.3×
ZeroClaw	77.8	13	18	1.1×
Four-system union	**88.4**	16	–	–

C.1.2 PAIRED OUTCOMES ON INTERACTIVE TASKS

On Qwen/Terminal-Bench 2, terminus-2 solves 35/89 tasks, OpenClaw 26/89, and ZeroClaw and OpenCode 13/89 each. Against the 88 paired terminus-2 outcomes, OpenClaw adds seven correct tasks and loses 16. With Gemma, the four-system union covers 25/89 tasks. Exact McNemar tests on 86 paired tasks give $p = 0.021$, 0.001, and 0.115 for OpenCode, ZeroClaw, and OpenClaw versus terminus-2.

C.1.3 ANSWER-EXTRACTION ABLATION

Tab. 7 re-scores the saved answer-based evaluations after removing the permissive extractor's final-line and final-integer fallbacks. Because the ablation only re-scores saved outputs, native execution remains fixed.

Table 7: **Framework-level ablation of answer extraction on the saved answer-based evaluations.** Each entry is the percentage-point change from the reported score after removing the permissive extractor's final-line and final-integer fallbacks and retaining only boxed or explicitly marked answers. Model outputs, task references, and scorers are unchanged; zero means that strict and reported extraction produce the same aggregate score. Green/red cells denote positive/negative changes under strict extraction.

Model	System	IMO-AnswerBench	HLE-Verifiable	GPQA-Diamond	AIME 26		MMMU-Pro
		Avg@1	Avg@1	Avg@1	Pass@4	Avg@4	Avg@1
Qwen3.5	Direct	0	0	0	0	0	0
	OpenClaw	−1.25	0	0	0	0	0
	ZeroClaw	−2.50	0	0	0	0	0
	OpenCode	−4.00	0	0	0	−0.83	0
Gemma-4	Direct	0	0	0	0	0	0
	OpenClaw	−0.25	0	0	0	0	0
	ZeroClaw	0	0	0	0	0	0
	OpenCode	0	0	0	0	0	0
Kimi-K2.6	Direct	−1.00	0	0	0	0	0
	OpenClaw	−1.25	0	0	0	−1.67	0
	ZeroClaw	0	0	0	0	0	0
	OpenCode	+0.25	0	0	0	−0.83	−0.12

C.2 Error Patterns in Failed Attempts

Analysis sample. Fig. 6 summarizes 2,857 recorded incorrect attempts: 1,108 from Gemma, 839 from Kimi, and 910 from Qwen. All 48 model–benchmark–system combinations are represented, with at least 30 attempts in 40 combinations. Sampling supplements sparse combinations independently of their error labels. Correct and unscored attempts, empty or corrupted records, and known scoring or task-matching defects are excluded. Rates describe the analyzed sample rather than full-archive prevalence.

Annotation procedure. GPT-5.6-sol with high reasoning effort assigns binary error labels in a single annotation stage, using the problem, saved trajectory and output, reference answer, and recorded grading result. Positive labels require supporting source references; accepted annotations pass schema and source-reference checks. Incomplete annotations are excluded from the analysis. For 255 MMMU-Pro attempts, missing images in the annotation packets are supplied from a pinned benchmark snapshot after matching task IDs, ordered options, and reference answers. These images provide retrospective task context, not evidence of the original model's visual access.

Error codes and counting. We report 17 labels in five families: Specification and State Design (SD), Reasoning and Control (RC), Tool and Evidence Interaction (TE), Verification and Finalization (VF), and Resource and Operational Reliability (OR). Tabs. 8–10 give each label's definition, required evidence, and decision boundaries. Unlike the six functional step types in Tab. 14, these labels identify errors supported by the recorded evidence. Each code receives *YES* for a supported incident and *NO* otherwise. Multiple labels may apply to one attempt; they are not mutually exclusive stages.

For group g and code c, incidence is $100\, n_{g,c}^{\text{yes}} / N_g^{\text{incorrect}}$. Each attempt counts at most once per code, including an error later recovered within that incorrect attempt. Codes overlap, so percentages need not sum to 100%. Fig. 6 pools attempts across the three models within each benchmark–system group. Repetition incidence does not measure cycle length or distinguish routine restatement from pathological loops; budget exhaustion is assessed separately from termination evidence.

Repetition across models. On HLE-Verifiable, OpenCode has a lower repetition rate than ZeroClaw within each model: 19.0% versus 33.3% for Gemma (15/79 versus 31/93), 16.2% versus 24.4% for Kimi (12/74 versus 19/78), and 66.0% versus 76.6% for Qwen (66/100 versus 36/47). Combining the models gives 93/253 (36.8%) for OpenCode and 86/218 (39.4%) for ZeroClaw. Different model proportions narrow the combined difference, which is smaller than the difference within any of the three models.

Sensitivity checks. Adding 49 annotations without source references gives 2,906 attempts and changes every overall code rate by less than 0.54 percentage points. Excluding the 255 image-

supplemented attempts gives RC-1/RC-4/VF-2 rates of 82.9%/71.0%/59.5%, compared with 83.2%/67.1%/57.5% in the main analysis; this comparison also removes four model–benchmark–system combinations. Equal weighting of all 48 combinations gives 82.4%/63.1%/53.0%. The ordering of these three leading labels is unchanged under both checks. Execution-level labels, source hashes, group counts, and plotting code accompany the analysis.

Definitions and decision boundaries. Tabs. 8–10 define the 17 labels in Fig. 6. Codes identify labels in the saved annotations; the figure uses readable names. Family names organize observable error indicators, not root-cause attributions. A positive judgment requires the stated evidence. An incorrect final answer alone is insufficient to infer a particular process error, and missing evidence is not proof that the behavior was absent.

Table 8: **Error definitions: specification and reasoning.** Row names match Fig. 6.

Code and label	Definition and required evidence
Specification and State Design (SD)	
SD-1: Task contract	Violates an explicit task requirement or output contract. Identify the requirement and conflicting action or output; a wrong answer or reasoning error alone does not qualify.
SD-2: Capability / interface	The exposed or withheld capability is materially mismatched to the task or model-facing contract. Require the actual configuration and necessary capability; absence claims require a complete relevant inventory and no equivalent route. Choosing not to use an available suitable interface is not this error.
Reasoning and Control (RC)	
RC-1: Reasoning error	A visible factual, mathematical, logical, perceptual, or inferential step conflicts with task evidence or a verified reference. Localize the incorrect proposition or inference; a wrong final answer without an evidenced erroneous step is insufficient. Later correction does not erase the incident.
RC-2: Decision–action inconsistency	A stated decision, plan, or conclusion conflicts with a later action or submitted answer. Identify both events and establish that no intervening explicit revision explains the change.
RC-3: Reasoning drift	Shifts to an unrelated objective or sustained off-goal activity without a justified path to task progress. Show the task goal and the departure; relevant exploration that is later abandoned does not qualify.
RC-4: Repetitive cycles	Repeats substantially the same reasoning, query, or action without new evidence, meaningful input changes, or progress. Compare at least two cycles. Justified retries, appropriate polling, and independent verification are excluded.
RC-5: Premature commitment	Commits or stops while an essential obligation remains unresolved. Show the agent-controlled decision and the obligation; omission claims require the complete relevant window. Forced budget/runtime stops and a wrong answer alone do not qualify.

Table 9: **Error definitions: Tool and Evidence Interaction (TE).**

Code and label	Definition and required evidence
TE-1: Tool selection / omission	Selects a materially unsuitable operation, or omits an explicitly required, available tool. Show the mismatch; omission requires the tool obligation, complete inventory and observation window, and no equivalent action. A permitted direct solution is not an omission.
TE-2: Tool execution	A concrete invocation fails through malformed arguments, unavailable capability, permission denial, timeout, process failure, or environment incompatibility. Require a failed call/result or terminal error state; pending asynchronous work is not failure.
TE-3: Observation integration	Available, relevant tool or environment feedback is ignored, misread, contradicted, or not incorporated. Identify the observation and incompatible later reasoning/action. Correctly integrating a defective source is a different issue.
TE-4: Evidence quality	Relies without adequate qualification on external evidence with a demonstrable support, relevance, or provenance defect. Identify the source defect and its use in the conclusion; internal knowledge errors or a source's status alone do not qualify.
TE-5: Recovery	After recognized failure feedback, a recovery attempt does not address the precursor condition and leaves it unresolved. Identify the feedback, attempted remedy, and unresolved result. No recovery attempt, a successful remedy, or a justified alternative failing for a new reason does not qualify.

Table 10: **Error definitions: verification, finalization, and operational reliability.**

Code and label	Definition and required evidence
Verification and Finalization (VF)	
VF-1: Verification omission	Submits a checkable candidate without a necessary check, or omits a decisive condition. Require a complete relevant trace, a feasible check, and an explicit obligation, unresolved expressed uncertainty/contradiction, or prespecified checklist item. Merely not showing extra checking is insufficient.
VF-2: Faulty verification	A check of an existing candidate uses a demonstrably faulty method or verdict. Identify the candidate and subsequent checking step; distinguish checking from first-pass reasoning and from an absent check. Later repair does not erase the incident.
VF-3: Answer commit / format	An identifiable intended answer is not committed, formatted, preserved, extracted, or transmitted according to the scorer-facing contract. Localize the boundary defect independently of semantic correctness; a faithfully delivered wrong answer or missing output alone is insufficient.
Resource and Operational Reliability (OR)	
OR-1: Budget exhaustion	A run-level token, context, time, step, or cost limit stops or truncates required processing. Require an explicit limit signal, finish reason, timeout, or limit-reaching metric. Length alone and a timeout confined to one tool call do not qualify.
OR-2: Runtime disruption	A harness, gateway, sandbox, provider, process, artifact, or infrastructure fault disrupts execution or scoring. Require a concrete run-level status or fault record; an isolated tool error handled within a healthy run is not sufficient.

Optional reasoning-error subtypes. RC-1 may additionally identify task interpretation (misreading the goal or conditions), perception/reading (misreading supplied visual or textual content), knowledge/concepts (using an incorrect fact or domain rule), numerical/symbolic calculation (arithmetic, algebra, units, or symbol manipulation), or logic/modeling (invalid assumptions, formalization, inference, or proof). If the evidence does not support a reliable subtype, retain RC-1 alone. These subtypes do not add independent counts to Fig. 6.

Overlapping incidents. RC-1 concerns an erroneous reasoning step, TE-3 concerns integrating an available observation, and VF-2 concerns checking an existing candidate. They may coexist when the record supports each judgment; do not assign several labels merely to redescribe the same error.

Similarly, premature commitment requires evidence beyond an already identified tool or verification omission. Label the supported behavior and execution level rather than inferring a hidden cause.

C.2.1 Human Alignment of Error Labels

Study and sample. Two human annotators separately labeled saved trajectories using the error taxonomy. We compare their recorded judgments with the frozen single-pass GPT-5.6-sol annotations used in that study, restricting the comparison to the 17 parent codes reported here. The core sample comprises 60 trajectories selected by fixed-hash ranking within model–benchmark–harness strata from 4,566 readable executions recorded as failures. Five additional tool-error cases were targeted supplements and are not part of the primary comparison.

Retrospective task and outcome screening retains 42 core trajectories covering 40 underlying problems. The remaining cases comprise six correct or equivalent answers under documented task assumptions, one task–trajectory mismatch, two incomplete tasks, and nine unresolved outcomes. Screening addresses source validity rather than selecting for agreement; all original judgments are preserved. Correct final answers can still contain process errors, but these cases are outside the screened-failure comparison.

Comparison. We pair judgments on the same trajectory and code and compare the presence or absence of a positive error assignment. Missing or unassigned judgments are excluded, not imputed. The primary comparison contains 714 paired decisions per rater pair. Overall agreement and Cohen's κ are computed from pooled binary decisions, with each trajectory and code equally weighted. LLM–human results average the two human-reference metrics, rather than using a constructed consensus label.

Table 11: **Overall human alignment of error annotations.** Parentheses give 95% problem-cluster bootstrap intervals. LLM–human denotes the mean over two human references. Agreement is descriptive, not a test of equivalence.

Comparison	Agreement (%)	Cohen's κ
Human–human	86.8 (84.5–89.2)	0.472 (0.393–0.554)
LLM–human	88.0 (86.0–89.9)	0.538 (0.470–0.602)

The LLM's aggregate agreement is comparable to the observed inter-annotator agreement (Tab. 11). Reference-specific agreement is 84.5% with H1 and 91.5% with H2. Because negative assignments are frequent, Tab. 12 also reports positive-label precision, recall, and F1 with each human as the reference. Mean positive F1 is 0.608, compared with 0.548 between humans. These results describe agreement with the two raters, not accuracy against an adjudicated gold standard.

Table 12: **Positive-error agreement by human reference.** H1 and H2 are anonymized annotators. Parentheses give 95% problem-cluster bootstrap intervals.

Reference	Precision	Recall	Positive F1
H1	0.417 (0.327–0.509)	0.549 (0.451–0.644)	0.474 (0.385–0.558)
H2	0.733 (0.650–0.819)	0.752 (0.686–0.815)	0.743 (0.684–0.798)

Uncertainty and sensitivity. We use 10,000 percentile bootstrap resamples (seed 20260913), sampling underlying problems within benchmarks and keeping associated attempts, codes, and raters together. Both LLM–human comparisons use the same resamples before averaging. Thus, code decisions are not treated as independent observations. Restricting the analysis to 347 decisions passing the archived common-evidence screen gives 86.7% LLM–human and 85.0% human–human agreement. Including the nine unresolved outcomes gives 87.3% and 86.2%, respectively, without treating those outcomes as resolved.

Scope. This study evaluates the frozen error-labeling baseline on the selected sample. Exact historical evidence access, annotator expertise, and consultation were not fully logged. The results do not

separately validate subsequent prompt refinements, optional RC-1 subtypes, or every annotation in the larger corpus. No additional human judgments or adjudicated labels are introduced in this analysis.

C.3 Functional-Step Annotation and Aggregation

Fig. 7 compares 2,751 scored trajectories across two models, three benchmarks, four systems, and two outcomes (48 cells). Tab. 13 reports coverage. Eligible trajectories have substantive saved content and resolvable event order. Supplementation uses a fixed, label-independent rank to improve sparse-cell coverage, while retaining existing larger cells. Correct and incorrect groups follow the recorded scorer; unscored attempts are excluded.

Table 13: **Trajectory coverage for Fig. 7.** Entries give correct/incorrect counts pooled over Direct, OpenClaw, ZeroClaw, and OpenCode.

Model	GPQA-Diamond	HLE-Verifiable	IMO-AnswerBench	Total
Gemma-4-31B-IT	299/62	295/359	303/300	1,618
Kimi-K2.6	219/46	185/256	219/208	1,133
Total	518/108	480/615	522/508	2,751

Step construction and labeling. Here an *action*, or *functional step*, is one saved assistant-text segment or one logged tool call, not necessarily an entire message or an external operation. An adapter removes duplicate storage copies while preserving source-linked assistant text, saved reasoning, and logged tool calls in order. Text is split into natural paragraphs or list steps; each actual call is a separate target. System, user, and tool-result events provide context rather than additional targets. Empty-text ZeroClaw turns require consistent call records and linked results. Missing reasoning is not reconstructed.

`GPT-5.6-sol` at high reasoning effort assigns one label per fixed step in a single classification stage, using the task and saved textual context (Tab. 14). The classifier receives text only. Long inputs are partitioned at step boundaries or use a long-context deployment with the same prompt. Accepted outputs cover every assigned step exactly once and pass source-linked structural checks; retries do not add observations.

Table 14: **Functional-step labels.** Labels describe a step's role, independently of whether its content is correct.

Label	Definition and boundary
Understanding	Identifies or restates the task's goals, givens, constraints, or answer form.
Planning	Proposes a strategy or next operation without carrying it out.
Reasoning	Derives, calculates, applies knowledge, or develops a candidate solution.
Tool Use	Issues a logged call or directly incorporates returned information, including a failure; a proposed call alone is Planning.
Verification	Checks or revises an existing result or interpretation; first-time derivation is Reasoning, and an unsuccessful check still qualifies.
Finalization	Commits to a task-level answer or terminal inability to answer; an intermediate value or boxed expression alone is insufficient.

For genuinely mixed steps, the tie-break is Finalization $>$ Tool Use $>$ Verification $>$ Reasoning $>$ Planning $>$ Understanding, applied only to functions supported by the context. Actual tool-call targets always receive Tool Use, including denied calls. Each step receives exactly one functional label; the labels are categories, not a prescribed stage sequence. A Verification step may perform either a valid or an invalid check, and Tool Use does not imply successful execution. These functional labels therefore differ from the multi-label error judgments in Appendix C.2.

Aggregation and weighting. Panel (a) pools labeled-step counts within each model–benchmark–system–outcome cell and divides by that cell's total. Panel (b) counts adjacent labels within HLE trajectories, including self-transitions but never cross-trajectory edges, and normalizes each panel by its transition total. These summaries describe pooled recorded activity, giving longer trajectories greater weight. Under equal-trajectory weighting, Kimi/ZeroClaw's incorrect HLE Verification share is 3.4%, compared with 26.7% when pooling steps; one trajectory contributes 211 of 217 Verification steps. This comparison distinguishes concentrated checking within a long trajectory from a pattern shared across trajectories.

C.3.1 Human Alignment of Functional-Step Labels

Protocol. Three human annotators were assigned the same 120 fixed action spans and the six definitions in Tab. 14. Each span was shown with surrounding context; annotators classified the target span rather than solving the task again. For 60 spans, machine labels were hidden and annotators selected a category independently. For the other 60, a machine suggestion was visible and annotators could accept it or choose another category. We analyze these phases separately. GPT-5.6-sol with high reasoning effort classified each fixed span in a single call using the corresponding textual context, without receiving human labels. All 120 machine outputs were valid.

Primary comparison. The machine-hidden phase contains 176 human judgments: two annotators completed all 60 spans and one completed 56. Missing judgments are not imputed. A human majority requires at least two matching votes and strictly more than half of the available votes. This yields a reference for 55 spans; five without a majority are excluded only from the majority-reference comparison, not from pairwise analyses. The classifier agrees with the majority on 50 of 55 spans (90.9%; $\kappa = 0.868$).

Table 15: **Human alignment on fixed action spans with machine labels hidden.** Pairwise rows average the three rater-pair metrics. The majority-reference row uses only spans with a human majority; its reference and denominator differ from those of the pairwise rows.

Comparison	Agreement (%)	Cohen's κ
Human–human (pairwise mean)	63.2	0.470
LLM–human (pairwise mean)	76.0	0.644
LLM–human majority	90.9	0.868

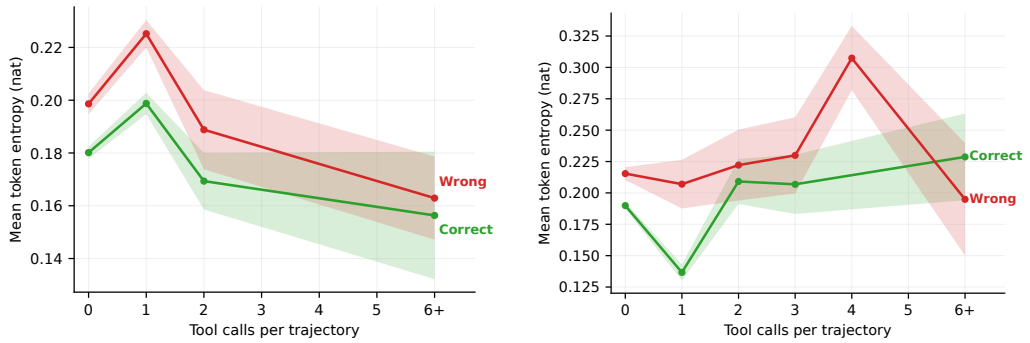
Agreement with the three individual annotators is 67.9%, 83.3%, and 76.7%, respectively. Restricting all comparisons to the 56 spans completed by all three annotators gives 75.0% mean LLM–human agreement and 62.5% mean human–human agreement. Thus, the pairwise pattern is not explained by the four missing judgments. The five disagreements with the majority concern boundaries among Reasoning, Planning, Verification, and Finalization.

Secondary comparison and scope. The machine-visible phase contains 174 human judgments. Mean LLM–human agreement is 67.2%, versus 67.7% between humans; majority-reference agreement is 73.2% on 56 spans. These judgments are machine-assisted and are not pooled with the primary comparison. The two phases use different spans, so their difference is not an estimate of the effect of showing a suggestion.

The annotation instructions were calibrated using this audit; these are descriptive calibration-set results, not held-out estimates of generalization. They assess six-way classification of fixed spans, not automatic step segmentation or full-trajectory annotation. Source-trajectory identifiers and population inclusion probabilities are unavailable for this audit, so we do not report population-level confidence intervals or significance tests.

C.4 Token-Entropy Patterns

Sec. 3 compares token entropy after tool calls and across full trajectories. Entropy is computed from the returned top-K token probabilities after renormalization, so it describes concentration within the recorded candidate set rather than the full vocabulary distribution. Lower entropy does not itself establish correctness.



(a) OpenClaw (b) OpenCode

Fig. 24: **Mean token entropy versus tool-call depth on MMMU-Pro (Gemma-4-31B-IT).** Each panel is one tool-using harness; the x-axis is the number of tool calls per trajectory and the y-axis is mean token entropy (nat). Green: correct; red: wrong; shaded bands are confidence intervals.

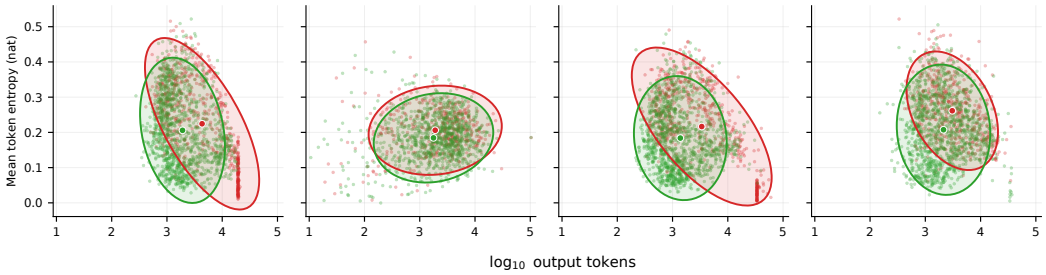


Fig. 25: **Trajectory-level mean token entropy versus output length on MMMU-Pro (Gemma-4-31B-IT).** Each point is one trajectory; green: correct; red: wrong, with per-outcome covariance ellipses and centroids. Length is on a $\log_{10}$ scale. From left to right, panels show *Direct*, OpenClaw, OpenCode, and ZeroClaw.

C.4.1 Entropy and Tool-Call Depth

Finding 11 (Fig. 24). Entropy trends with tool-call depth depend on the harness. Mean token entropy declines with depth in one harness but peaks at intermediate depths on wrong trajectories in the other.

Fig. 24 plots mean token entropy against the number of tool calls. In the left panel, entropy peaks at the first tool call and then falls steadily for both outcomes, which sit close together. This association does not isolate the effect of tool feedback. The right panel is the opposite: wrong-trajectory entropy rises with tool depth, peaks near four calls, and only at 6+ calls does the correct curve move above the wrong one. These are comparisons among trajectories with different tool-call counts, not an intervention that adds a call to the same trajectory; the plot alone cannot attribute the entropy differences to tool quality.

C.4.2 Entropy and Output Length

Finding 12 (Fig. 25). Confident long-tail failure persists here, but its entropy direction depends on the harness. Wrong trajectories form a long, low-entropy tail under the tool- and finalization-heavy harnesses, yet carry *higher* entropy than correct ones under the no-tool harness.

Fig. 25 plots trajectory-level entropy against output length. For *Direct* (leftmost panel) and OpenCode (third panel), the red ellipse tilts downward into the lower-right, and a vertical red streak pins many wrong trajectories at the length cap with low entropy: long, fluent, low-entropy chains that commit to a wrong reading and never recover, as in Finding 4. OpenClaw (second panel) is the least separable, with overlapping ellipses and short outliers from aborted tool loops. ZeroClaw (rightmost panel) flips the direction: its wrong centroid sits *above* the correct one, indicating a less concentrated recorded

next-token distribution on average. Mean entropy alone is thus not a reliable correctness signal: in long trajectories low entropy means confident commitment to a wrong answer, while under the ZeroClaw no-tool template wrong trajectories are the higher-entropy ones.

C.5 Token Use and Wall-Time Sensitivity

Appendix B.2 specifies output-token accounting and the observed task sets used in Fig. 9. The analysis below uses a separate matched-attempt intersection to examine realized time and timeout sensitivity.

The macro evaluation compares complete systems at one predeclared operating point rather than tuning a separate resource optimum for each harness. Matched conditions share the model context configuration and a per-request generation limit of `max_tokens=` 131,072. The controller or request cap is 9,300 seconds; OpenClaw additionally uses a 9,000-second stream-total cap, while ZeroClaw limits each run to 100 tool iterations and 200 actions per hour. These safeguards were fixed before evaluation. Because multi-turn systems can terminate early or issue different numbers of requests, fixed limits bound the comparison but do not equalize realized compute.

To measure realized time on matched work, we use nine validated model–benchmark blocks spanning all three models and four answer-based benchmarks. Within each block, we intersect attempt IDs across *Direct*, OpenClaw, OpenCode, and ZeroClaw, retain failed and timed-out attempts, compute each system's mean, median, and 90th-percentile wall time, and then macro-average each statistic across blocks. This yields 6,848 matched attempts per system. Tab. 16 shows substantial long-tail latency, but no monotonic relation between longer realized time and higher accuracy.

Table 16: **Macro-averaged realized wall time over matched attempts.** Statistics are computed separately in nine model–benchmark blocks and then macro-averaged. Each system contributes 6,848 matched attempts, including failures and timeouts.

System	Mean (s)	Median (s)	p90 (s)
Direct	682.7	429.2	1,124.2
OpenClaw	576.8	265.4	1,635.4
OpenCode	530.9	217.1	1,450.2
ZeroClaw	513.3	259.3	1,189.5

We next re-score the same attempts at 17 lower wall-time caps up to 9,300 seconds, counting an otherwise correct attempt as unsuccessful when it completes after a cap. In 35 of 36 model–system–benchmark cells, some smaller cap retains at least 99% of the original accuracy. The exception is Qwen3.5-27B/GPQA-Diamond/OpenCode, which remains right-censored at 9,300 seconds. After all four systems in a block cross the 99% threshold, their ordering remains unchanged in all eight blocks with at least two larger cap values. The sweep therefore rules out a simple explanation in which the reported rankings arise from allowing one system to run longer within the tested range.

Wall time combines model-serving latency, harness overhead, tool execution, and early termination. The lower-cap sweep measures the sensitivity of the reported rankings to AlphaDiana's timeout policy within the tested range.

D COMPONENT INTERVENTION PROTOCOLS

This appendix follows Sec. 4: matched configurations, tool removal, skill loading, memory persistence, and costs across full harness configurations. Shared evaluation settings appear in Appendix B.

D.1 SINGLE-COMPONENT CONFIGURATION MATRIX

Table 17: **Configuration matrix for the single-component interventions.** ✓/✗ indicate whether the named entry point is active, and the shaded cell marks the only change from *Default Harness*. The default configuration retains its native memory interface but does not explicitly elicit memory use. Within each matched comparison, the agent loop, model, benchmark, scorer, decoding parameters, and sampling budget are fixed.

Configuration	Tool registry	Skill library	Memory elicitation
Default Harness	✓	✗	✗
Tool Registry Removed	✗	✗	✗
Skill Loaded	✓	✓	✗
Memory-Elicited	✓	✗	✓

D.2 TOOL REGISTRY REMOVAL

Tab. 18 specifies the native tool registries removed by the tool-registry intervention and the procedural libraries mounted by the skill-loading intervention. These contents are fixed within each matched intervention in Sec. 4. Tool removal suppresses the selected tool descriptions and related prompt instructions while retaining the matched model, task, scorer, decoding, budget, and agent loop. The skill-library rows describe the separate loading intervention in Appendix D.3.

Table 18: **Tool surfaces and procedural libraries used in the component interventions.**

Component	Contents
OpenCode v1.3.2	Shell, file operations and search, task execution, web retrieval, task lists, and skill loading
ZeroClaw v0.6.9	Shell, file operations and search, tasks, web, scheduling and memory, configuration, version control, calculation, weather, and canvas
Math library	A seven-step quantitative protocol with references for major mathematical domains
General library	17 skills covering documents, UI, design, communication, APIs, and web development and testing

D.3 SKILL LOADING

Libraries are mounted through each harness's native interface, not inserted into every request; native tools remain available. Loading adds a library pointer. OpenCode matches loading-instruction wording across libraries; the plotted ZeroClaw runs mandate Math loading but make General loading optional. Thus, Fig. 11 measures the complete loading configuration, and read-rate differences do not isolate library content from its advertised identity or loading instruction. Fig. 12 counts a read only when a trajectory opens the top-level `SKILL.md`.

D.4 MEMORY MECHANISMS AND PERSISTENCE

The memory study uses Qwen3.5-27B on AIME 26 with fixed model weights. OpenCode includes prior solutions in the prompt for later attempts. OpenClaw and ZeroClaw store summaries and retrieve notes through semantic search. Raw interaction histories and workspace files are reset between attempts. Only the designated memory is retained, with no separate answer cache.

The three settings differ in how long this memory is kept:

- **Cross-Rollout.** Four attempts on a task share writable memory, cleared before the next task. These attempts are dependent because later ones can use earlier solutions.
- **Cross-Task.** Four separate runs each process Tasks 1–30 in the same order. Each run starts with empty memory and keeps it across tasks. The four runs do not share memory.
- **Cross-Dataset.** Memory is built on AIME 25 and frozen before AIME 26. Attempts on AIME 26 read this memory without updating it.

Across these three settings, the task prompt, memory instruction, limits, resources, and sampling are matched. *Default Harness* does not explicitly instruct the agent to use memory and clears memory between attempts. Comparing it with *Memory-Elicited* therefore measures the combined effect of memory instructions and persistence. Comparisons across harnesses cannot isolate summary quality, retrieval, prompt length, or control logic as the cause of the score differences.

D.5 Costs Across Full Harness Configurations

The final cost comparisons in Sec. 4 compare full harness configurations rather than isolating an individual component. Figs. 15 and 16 use the common observed Gemma task sets and output-token accounting specified in Appendix B.2. The cost per solved attempt includes tokens spent on unsuccessful attempts; the high-token and unsuccessful-attempt shares overlap. These comparisons therefore describe the cost of obtaining correct answers under each complete configuration, not the causal overhead of a particular tool, skill, or memory mechanism.

E Related Work

This work studies *system-level* evaluation of agentic reasoning. Our starting point is that an open agent is best viewed not as a model alone, but as a *model-harness system*: the *model* provides raw generative and reasoning capacity, while the *harness* determines how that capacity is realized through control flow, tool access, context management, memory, interaction protocols, runtime policies, and feedback handling. This perspective connects several research threads: reasoning-capable foundation models, agent runtimes and tool-use frameworks, memory and context-management mechanisms, benchmark families ranging from model-centric reasoning tasks to interactive agent environments, and evaluation protocols that emphasize trajectories, robustness, and efficiency. It also clarifies a key distinction. *Model reasoning* studies a standalone model under direct inference, whereas *agentic reasoning* studies a model embedded in a multi-step loop that can act, observe, revise, and continue across turns. Once reasoning is coupled with action, environment interaction, and persistent state, the object of evaluation is no longer just the model, but the full system that turns the model into an agent. We therefore organize related work around the transition from *model-only* evaluation to *model-harness* evaluation. The system interfaces in Appendix A, macro analyses in Appendix C, and component protocols in Appendix D provide the concrete setting for this comparison.

E.1 Reasoning Foundation Models

What this line of work evaluates. A large body of work evaluates the reasoning ability of language models under direct or lightly structured inference. Chain-of-thought prompting showed that eliciting intermediate reasoning steps can substantially improve complex reasoning in large models (Wei et al., 2022), while benchmarks such as MATH and GSM8K became standard tests of mathematical reasoning (Hendrycks et al., 2021; Cobbe et al., 2021). Broader evaluation efforts such as HELM emphasized the importance of standardized, multi-metric, and scenario-diverse assessment of language models (Liang et al., 2022), while challenging reasoning-focused suites such as BIG-Bench Hard highlighted persistent gaps in compositional and symbolic reasoning even for strong language models (Suzgun et al., 2022). Subsequent work explored inference-time search and aggregation strategies, including self-consistency and tree-based reasoning, showing that measured reasoning quality depends not only on the base model but also on the inference procedure used to query it (Wang et al., 2023; Yao et al., 2023a). Open-weight reasoning models such as Qwen3 and DeepSeek-R1 further demonstrate that targeted post-training and reinforcement learning can substantially improve reasoning performance beyond standard instruction tuning (Yang et al., 2025; DeepSeek-AI et al., 2025). In parallel, API-based reasoning models expose explicit deliberation or "thinking" modes and have become common baselines for complex reasoning and agentic workflows (OpenAI, 2024; OpenAI; Google).

Why this is still insufficient for harness-aware evaluation. These works are primarily evaluated at the *model level*, usually under direct prompting, fixed sampling budgets, and standardized inputs. That setting is appropriate when the target of study is the standalone model. However, in realistic use, strong reasoning increasingly depends on how a model is embedded in a runtime with tools, memory, observations, retries, and execution policies. As a result, this literature tells us a great deal about *raw model reasoning*, but much less about how reasoning changes once the model is operationalized by a harness. This creates a gap between *model reasoning* as it is usually benchmarked and *agentic reasoning* as it is actually deployed.

E.2 Agentic Reasoning Systems and Open Agents

What this line of work evaluates. Modern reasoning systems increasingly combine a foundation model with planning, tool use, memory, and environment interaction, producing long-horizon trajectories rather than single completions. Foundational examples include ReAct (Yao et al., 2023b), which interleaves reasoning and acting, LATS (Zhou et al., 2024), which performs search over action trajectories, and Toolformer, which trains models to invoke tools in a self-supervised manner (Schick et al., 2023). More broadly, recent work on externalized reasoning emphasizes that capabilities often emerge from structured interaction between a model and external resources rather than from the parametric model alone (Zhou et al., 2026a).

Open agent systems vary widely in how they realize this interaction. **Gateway- and runtime-centered** systems treat sessions, routing, and persistent state as first-class concerns; for example, OpenClaw embeds a model in a sessionful gateway with tool interfaces, memory, and policy controls. **Tool-oriented** frameworks emphasize modular tool abstractions, planning, and execution, aiming to improve reasoning through better external affordances rather than extra model training (Lu et al., 2025b). **Domain- and workflow-oriented** systems, especially in scientific settings, prioritize execution grounding and traceability (Zhang et al., 2025a). AlphaApollo similarly orchestrates multiple models and professional tools through explicit iteration and structured tool use (Zhou et al., 2025b). Recent surveys synthesize this broader shift from model-centric reasoning toward autonomous, lifelong, and self-improving agent systems (Zheng et al., 2025a; Ferrag et al., 2025).

Why this is still insufficient for harness-aware evaluation. This literature establishes that many agent capabilities arise from the surrounding runtime rather than from the base model alone. But it typically studies complete systems, methods, or design patterns, rather than isolating the harness as an explicit object of evaluation. As a result, two agents built on the same model may behave very differently because of differences in orchestration, tool routing, state management, or feedback handling, yet those differences are often reported only as properties of the overall system. For our purposes, this means the key missing step is not another agent architecture, but a framework that can compare *matched model-harness systems* and attribute performance differences to the harness itself.

Memory and context management. A central source of variation across agent systems is how they manage context windows and persistent state. Early work on long-context efficiency focused on compression and memory virtualization, for example LongLLMLingua for prompt compression (Jiang et al., 2024) and MemGPT for OS-inspired hierarchical memory management (Packer et al., 2023). Subsequent work introduced retrieval- and structure-based long-term memory, including HippoRAG (Gutiérrez et al., 2024), temporal knowledge-graph memory in Zep (Rasmussen et al., 2025), object-centric persistent memory (Zahn & Chana, 2026), graph-native versioned memory architectures (Park, 2026), provenance-aware verified memory (Zhu et al., 2026), and governance-oriented memory control (Kumar et al., 2026). A second line of work studies working-memory management for long-horizon agents, including hierarchical working memory in HiAgent (Hu et al., 2025), agentic memory modules such as A-MEM and Mem0 (Xu et al., 2025; Chhikara et al., 2025), adaptive memory distillation and memory-structure selection (Ma et al., 2025; Lu et al., 2026), query-aware budget-tier routing (Zhang et al., 2026b), and compression or pruning methods such as ACON, Active Context Compression, SWE-Pruner, and trajectory reduction (Kang et al., 2025; Verma, 2026; Wang et al., 2026; Xiao et al., 2026). Other work emphasizes evolving contexts and summarization-based management, including COMPASS, context folding, and end-to-end summarization for multi-turn RL (Wan et al., 2025; Sun et al., 2025; Lu et al., 2025a), as well as efficient lifelong and self-evolving memory systems such as Dynamic Cheatsheet, SimpleMem, MemRL, MemSkill, and General Agentic Memory (Suzgun et al., 2026; Liu et al., 2026b; Zhang et al., 2026c;a; Yan et al., 2025).

What this line of work evaluates is how different memory mechanisms improve long-horizon reasoning, retrieval, persistence, or adaptation. *Why this is still insufficient* is that memory is usually treated as a design improvement inside a particular agent, rather than as a controlled harness factor whose contribution can be measured under matched models, tasks, and budgets. For our framing, memory is not merely a model property; it is a harness choice that can substantially alter performance even when the base model is fixed.

Routing, orchestration, planning, and budgeted control. Another major aspect of the harness concerns how computation is allocated and how decisions are organized across steps, tools, and sub-agents. Planning-oriented approaches such as Plan-and-Act improve long-horizon behavior through explicit planning before execution (Erdogan et al., 2025). Recent work extends this into broader orchestration and adaptive control, including reflective prompt evolution in GEPA (Agrawal et al., 2025), generalized routing across models and agents (Guo et al., 2025), evolving contexts for self-improving agents (Zhang et al., 2025b), orchestration frameworks such as Orchestral AI (Roman & Roman, 2026), context-aware multi-agent routing in CASTER (Liu et al., 2026c), intention-based planning under costly tool budgets (Liu et al., 2026a), and adaptive routing via active inference in ODAR (Ma et al., 2026).

What this line of work evaluates is how better control policies improve long-horizon behavior, budget allocation, or tool-use efficiency. *Why this is still insufficient* is that these methods usually optimize a specific system rather than support controlled comparison across different harnesses under a shared protocol. Together, these works show that the harness is not a passive wrapper but an active policy layer that determines planning depth, routing decisions, budget allocation, and fallback behavior. What remains underexplored is how much these factors contribute to measured reasoning performance when the underlying model is held fixed.

Skills, tool abstraction, and compositional reasoning. A related line of work studies *skills* as higher-level abstractions that sit between a model and raw tool calls. Examples include induced programmatic skills for agentic tasks (Wang et al., 2025b), compositional skill synthesis for reasoning (Jiao et al., 2026), empirical analyses of deployed skill ecosystems (Ling et al., 2026), lifelong skill self-evolution (Yang et al., 2026b), and token-efficient skill optimization (Gao et al., 2026).

What this line of work evaluates is how structured action abstractions can make agent behavior more effective or efficient. *Why this is still insufficient* is that skills reshape the effective action space and control flow seen by the model, yet are rarely analyzed as part of a broader harness decomposition. These abstractions therefore naturally belong to the harness layer rather than to the base model alone, and should be evaluated as such.

Self-improving and self-evolving agents. The same is true for systems that adapt at test time through experience, memory updates, tool-policy updates, or self-modification. Recent work studies EvoAgent (Feng et al., 2025), self-evolving multi-agent simulation frameworks (Almansoori et al., 2025), SafeSieve for experience-driven pruning in multi-agent communication (Zhang et al., 2025c), EvoTool for self-evolving tool-use policies (Yang et al., 2026a), and Memento, which improves agents by tuning the surrounding system rather than the base model itself (Zhou et al., 2025a). Test-time learning is also explored through adaptive memory and human guidance, as in Dynamic Cheatsheet and human-in-the-loop self-improvement methods (Suzgun et al., 2026; He et al., 2025), while Evo-Memory makes self-evolving memory an explicit evaluation target (Wei et al., 2025b).

What this line of work evaluates is how agents can improve during deployment through adaptation of memory, policy, or surrounding scaffolds. *Why this is still insufficient* is that these gains are often reported as improvements in the overall agent, without disentangling whether the change comes from better reasoning, better adaptation mechanisms, or more favorable harness policies. These works reinforce the same point: gains in agent performance often arise through harness adaptation rather than parametric model changes alone.

Harness engineering, runtime infrastructure, and observability. A recent and rapidly growing body of work studies the harness layer itself as a systems problem. Some work focuses on observability, proposing system-level tracing and structured logs for agent execution, such as AgentSight and AgentTrace (Zheng et al., 2025b; AlSayyad et al., 2026). Other work studies governed or typed execution, hierarchical retrieval interfaces, or infrastructure for complex codebase interaction, for example POLARIS, A-RAG, and Codified Context (Moslemi et al., 2026; Du et al., 2026; Vasilopoulos, 2026). There is also growing interest in automatically constructing or optimizing the harness, including AutoHarness, domain-specific harnesses for terminal coding and compilers, and explicit optimization of model-harness combinations in Meta-Harness (Lou et al., 2026; Bui, 2026; Zheng et al., 2026; Lee et al., 2026). Recent conceptual work makes this systems view explicit by treating harness engineering as a distinct layer of control, runtime, and agency for language agents (He et al., 2026).

What this line of work evaluates is the harness as a systems artifact: something that can be instrumented, governed, synthesized, and optimized. *Why this is still insufficient* is that most of this work still does not provide a general evaluation substrate for matched comparison of heterogeneous model-harness systems on shared tasks with shared verifiers and budgets. If harnesses can be traced, governed, synthesized, and optimized as modular artifacts, then they should also be evaluated as such.

Security and safety. Because agent behavior is mediated by execution environments, tools, memory, and runtime policies, security and safety arise at the harness level rather than purely at the model level. Recent work on OpenClaw emphasizes security controls in session-based tool execution, while

broader studies analyze how to secure agent execution under realistic runtime threats and failure modes (Bühler et al., 2025).

What this line of work evaluates is whether runtime policies, containment, and execution safeguards can make agents safer in practice. *Why this is still insufficient* is that safety-oriented harness choices can also affect measured task performance, making it difficult to interpret end-to-end scores unless policy enforcement and reasoning ability are considered together. This perspective reinforces the need for evaluation settings that can separate raw reasoning ability from policy enforcement, containment, and execution safeguards.

E.3 Reasoning and Interactive Evaluation Benchmarks

Benchmarking work relevant to our setting falls into two broad categories: **model-centric reasoning benchmarks** and **interactive agent benchmarks**. This distinction mirrors the transition studied in our paper: from evaluating a standalone model under direct inference to evaluating a model embedded in a harness that can act, observe, and interact with an environment.

Model-centric reasoning benchmarks. A large body of work evaluates language models on mathematical, scientific, code, and formal reasoning under a prompt-in/answer-out protocol. Representative examples include MATH (Hendrycks et al., 2021), GSM8K (Cobbe et al., 2021), GPQA (Rein et al., 2024), SciBench (Wang et al., 2024), BBH (Suzgun et al., 2022), HumanEval (Chen et al., 2021), and LiveCodeBench (Jain et al., 2024). Broader model-evaluation efforts such as HELM also highlight the importance of standardized evaluation across scenarios and metrics (Liang et al., 2022). More recent frontier and verifier-backed benchmarks extend this paradigm while retaining relatively clean correctness criteria. Humanity's Last Exam is designed to remain difficult at the knowledge frontier while preserving closed-ended grading (Phan et al., 2025), and HLE-Verified was introduced to improve reliability and reduce ambiguity through structured verification (Zhai et al., 2026). FrontierMath similarly emphasizes advanced mathematical reasoning with automated verification to reduce ambiguity and contamination (Glazer et al., 2024), while FormalMATH targets formal mathematical reasoning in theorem-proving-style settings (Yu et al., 2025b). MMMU-Pro evaluates multi-discipline multimodal understanding and reasoning (Yue et al., 2024).

What this line of work evaluates is raw reasoning capability under controlled inference and relatively reliable judging. *Why this is still insufficient* is that most model-centric reasoning benchmarks assume a fixed inference protocol and evaluate the model as the primary object of interest. They are therefore ideal for measuring *raw model reasoning*, but do not by themselves distinguish model capability from harness capability once reasoning is mediated by tools, memory, execution support, and multi-step interaction.

Interactive agent benchmarks. A second line of work evaluates agents through action in realistic environments rather than through answer matching alone. Early and influential examples include AgentBench (Liu et al., 2024), Mind2Web (Deng et al., 2023), WebArena (Zhou et al., 2023), VisualWebArena (Koh et al., 2024), and WebVoyager (He et al., 2024), which study web interaction, browsing, and multimodal web-based tasks. Other benchmarks target broader forms of digital and workplace interaction, including OSWorld for open computer-use tasks (Xie et al., 2024), WorkArena for common knowledge-work tasks (Drouin et al., 2024), and GAIA for general AI assistants solving realistic multi-step tasks (Mialon et al., 2023). A large coding- and software-oriented line includes SWE-bench for resolving real GitHub issues (Jimenez et al., 2024), Terminal-Bench for shell-based workflows (Merrill et al., 2026), ToolSandbox for stateful conversational tool use (Lu et al., 2024), R2E-Gym for procedural software-engineering environments with hybrid verifiers (Jain et al., 2025), ProdCodeBench for production-derived coding-agent evaluation (Jha et al., 2026), and FeatureBench for complex feature-level agentic coding tasks (Zhou et al., 2026b). Additional benchmarks emphasize realistic assistant, research, or collaborative-agent settings, including AssistantBench (Yoran et al., 2024), BrowseComp (Wei et al., 2025a), CORE-Bench for computational reproducibility (Siegel et al., 2024), DeepResearchGym (Coelho et al., 2025), ResearchGym (Garikaparthi et al., 2026), MLGym (Nathani et al., 2025), InnovatorBench for AI research agents (Wu et al., 2026), MultiAgentBench for collaboration and competition among multiple agents (Zhu et al., 2025a), and Agentified Benchmarking for Logical Reasoning Agents (Ni et al., 2026). Newer multi-turn and workflow-oriented settings such as τ-bench, WebWalker, NESTFUL, and τ-Knowledge place even greater

emphasis on structured tool use, conversational state changes, and policy-constrained interaction (Yao et al., 2024; Wu et al., 2025; Basu et al., 2024; Shi et al., 2026). Recent work also argues that sufficiently rich computer environments can elicit broader forms of agentic intelligence than static prompting alone (Cheng et al., 2026).

What this line of work evaluates is end-to-end behavior of agents acting in realistic environments. *Why this is still insufficient* is that end-to-end performance depends not only on the base model, but also on the surrounding harness, including tool interfaces, control flow, observation formatting, memory and state handling, and runtime policies. This makes interactive agent benchmarks highly valuable for measuring full-system behavior, while also motivating harness-aware evaluation frameworks that can better separate model effects from harness effects.

Taken together, these two benchmark families motivate the core distinction in our paper. *Model-centric reasoning benchmarks* evaluate a model in isolation under direct inference, whereas *interactive agent benchmarks* evaluate a model-harness system acting in an environment. Our empirical design focuses on largely verifiable reasoning settings because they provide a clean testbed for connecting these two traditions: they retain reliable correctness signals from model-centric evaluation while surfacing the system-level effects that arise once a model is embedded in a harness.

E.4 Evaluation Protocols, Reliability, and Human–Agent Interaction

What this line of work evaluates. A growing body of work argues that agent evaluation cannot be reduced to final accuracy alone. Agent behavior is shaped by planning quality, tool choice, policy compliance, recoverability, robustness, efficiency, and interaction quality. AgentBench evaluates LLMs as agents across diverse environments and highlights gaps in long-horizon reasoning and instruction following (Liu et al., 2024). T-Eval evaluates tool use *step by step* rather than only through final outcomes (Chen et al., 2024). τ-bench proposes reliability-oriented metrics over repeated trials in dynamic tool-agent-user interactions (Yao et al., 2024). MCP-Bench emphasizes trajectory-level planning and cross-tool coordination through standardized tool servers (Wang et al., 2025a). OSWorld-Human complements success metrics with human reference trajectories and efficiency diagnostics (Abhyankar et al., 2025). AgentRewardBench further studies the reliability of automatic trajectory evaluation itself, focusing on how web-agent trajectories should be judged rather than only how agents should act (Lù et al., 2025).

Recent work expands this picture in several directions. Some papers focus on debugging and recoverability, including failure analysis for agent trajectories and explicit metrics such as ERR (Zhu et al., 2025b; Vuddanti & Chittiprolu, 2026). Others study robustness under more realistic interfaces, such as imperfect APIs, context retrieval in coding agents, and memory stress tests in long-horizon interactions (Kim et al., 2026; Li et al., 2026; Hu et al., 2026; Jiayang et al., 2026). A broader reliability perspective is developed in recent work calling for a more systematic science of AI agent reliability, alongside practical regression-testing and readiness frameworks such as AgentAssay and LLM Readiness Harness (Rabanser et al., 2026; Bhardwaj, 2026; Maiorano, 2026). Related work also highlights evaluation variance and protocol sensitivity, for example through analyses of randomness in agentic evaluation (Bjarnason et al., 2026) and through more rigorous evaluation protocols for coding agents on SWE-bench (Yu et al., 2025a). Related work also studies when human collaboration is preferable to full automation and how human feedback can improve agent behavior at test time (Natarajan et al., 2024; He et al., 2025).

Why this is still insufficient for harness-aware evaluation. Collectively, these protocols motivate trajectory-aware logging, reproducibility mechanisms, modular judging, and reliability analysis beyond a single scalar success metric. However, most of this work aims to improve how end-to-end agents are evaluated, rather than to isolate what portion of measured performance should be attributed to the harness itself. In other words, it strengthens the measurement of agent behavior, but does not by itself resolve the attribution problem created when the evaluated object is a model embedded in a complex runtime.

E.5 From Model-Only to Model–Harness Evaluation

What remains missing. Despite substantial progress on reasoning models, memory systems, routing and orchestration, skill abstractions, benchmark suites, reliability analyses, and runtime

infrastructure, the field still lacks a unified **system-level evaluation substrate** that composes heterogeneous agents, benchmark drivers and judges, and execution backends under controlled policies while producing replayable trajectories. In practice, evaluation code is often tied to a specific benchmark harness, runtime, or tool protocol, making cross-system comparison brittle and difficult to reproduce. This creates a persistent attribution problem: the same model may appear stronger or weaker depending on the surrounding scaffold, while end-to-end benchmark scores often conflate model capability with orchestration quality, context management, or execution policy.

How our work positions itself. Our work targets this gap directly. We seek to move from *model-only* evaluation to *model-harness* evaluation by treating the agent as the object of study at the macro level while also analyzing the harness as a decomposable source of performance differences at the micro level. In particular, we focus on open agents evaluated in controlled, largely verifiable reasoning settings, where correctness can be checked deterministically and harness effects can be studied more cleanly. AlphaDiana is designed as a unified substrate for this purpose: it provides common abstractions for agent adapters, benchmark drivers, sandbox backends, isolated judges, and trajectory-level observability, enabling controlled comparisons such as "Qwen3-8B + OpenClaw on HLE" without entangling benchmark logic with agent integration. This goal is closely aligned with broader calls for modular, transparent, and reproducible evaluation infrastructure in both model-centric and agentic settings (Liang et al., 2022; Kapoor et al., 2026). At the same time, recent benchmarks for AI research, logical reasoning agents, coding agents, and reproducibility-oriented agents further illustrate that the community is rapidly expanding what counts as an agent evaluation target (Siegel et al., 2024; Garikaparthi et al., 2026; Wu et al., 2026; Jha et al., 2026; Zhou et al., 2026b; Ni et al., 2026). What remains underdeveloped is a framework for *matched* comparison across open model-harness systems, where the model, harness, benchmark, verifier, and budget policy can be varied in controlled ways. In this sense, AlphaDiana is not merely another benchmark harness, but an infrastructure layer for systematic *model-harness* evaluation.