OPENCOMPUTER: VERIFIABLE SOFTWARE WORLDS FOR COMPUTER-USE AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

We present **OpenComputer**, a verifier-grounded framework for constructing verifiable software worlds for computer-use agents in existing real desktop applications. OpenComputer integrates four components: (1) app-specific state verifiers that expose structured inspection endpoints over real applications, (2) a self-evolving verification layer that improves verifier reliability using execution-grounded feedback, (3) a task-generation pipeline that synthesizes realistic and machine-checkable desktop tasks, and (4) an evaluation harness that records full trajectories and computes auditable partial-credit rewards. In its current form, OpenComputer covers 33 desktop applications and 1,000 finalized tasks spanning browsers, office tools, creative software, development environments, file managers, and communication applications. Experiments show that OpenComputer's hard-coded verifiers align more closely with human adjudication than LLM-as-judge evaluation, especially when success depends on fine-grained application state. Frontier agents struggle with end-to-end completion despite partial progress, and open-source models exhibit sharp drops from their OSWorld-Verified scores, exposing a persistent gap in robust computer automation.

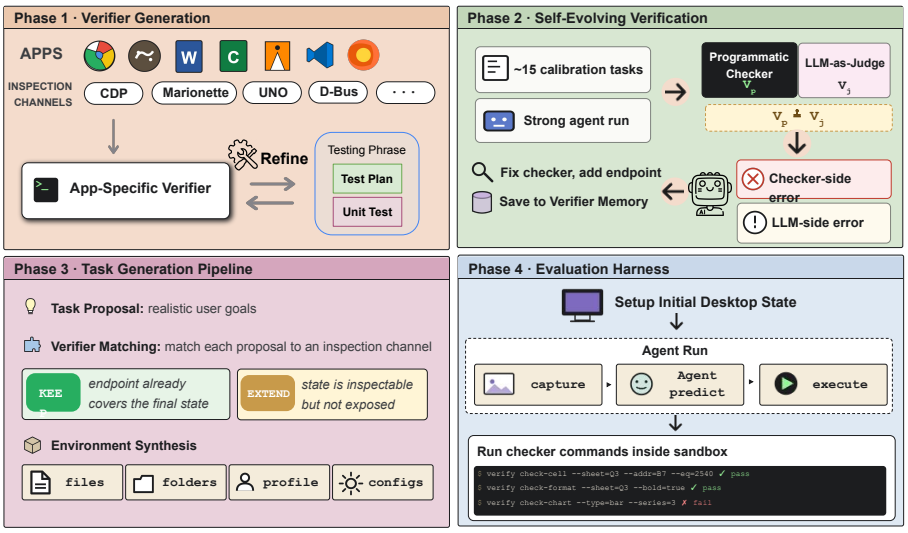


Figure 1: Overview of the OpenComputer pipeline: Phase 1 builds and tests app-specific verifier endpoints over reliable inspection channels. Phase 2 refines verifiers through execution-grounded disagreement analysis. Phase 3 synthesizes verifier-matched task instances with executable environments. Phase 4 runs agents in the sandbox and computes programmatic rewards.

1 INTRODUCTION

Computer-use agents offer a promising path toward general-purpose AI systems that operate the same software interfaces humans use every day (Agashe et al.; Nguyen et al., 2025; Agashe et al., 2025; Song et al., 2025a), but scaling their training and evaluation is limited by the cost of constructing realistic, reproducible desktop environments and tasks (Xu et al., 2024; He et al., 2024).

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.

Constructing a realistic desktop task involves far more than writing a natural-language instruction. A human developer must first design a plausible user goal, then manually prepare the underlying environment state (*e.g.,* creating or editing files, configuring folders, populating spreadsheets or documents, setting browser history or bookmarks, preparing emails or calendars), and ensures that the software state is both coherent and reproducible (Xie et al., 2024; Bonatti et al., 2024). These steps are tedious, application-specific, and difficult to standardize, making large-scale task creation slow and expensive.

Beyond environment construction, computer-use tasks also require trustworthy verification of the resulting software state. In desktop settings, success is often reflected not only in visible screenshots, but also in application state, file contents, metadata, or persistent side effects (Xie et al., 2024; Bonatti et al., 2024). This makes evaluation difficult to scale: each task often requires custom inspection logic that can determine whether the intended state has actually been achieved. A natural fallback is to use an LLM-as-a-judge (Liu et al., 2023; Kim et al., 2023), but this introduces substantial limitations. LLM judgments can be sensitive to prompt wording, incomplete observations, and model-specific biases, and are often difficult to audit or reproduce across runs (Wang et al., 2024; Li et al., 2025a; Thakur et al., 2025; Zheng et al., 2023). More importantly, an LLM judge may reward outcomes that appear plausible from screenshots while missing errors in the underlying software state (Sumyk & Kosovan, 2026; Cui et al., 2026). One way to simplify this problem is to synthesize application interfaces as web-based or reconstructed environments (Cao et al., 2026; Zhang et al., 2026), giving the environment designer direct control over the implementation, state representation, and interaction logic. While this controllability facilitates scalable environment construction, synthetic replicas are not guaranteed to preserve the full behavior of real desktop software, especially for feature-rich, stateful applications such as Blender, FreeCAD and darktable. For agents intended to operate real software, training and evaluation directly within existing applications are therefore essential for exposing them to the native workflows and state semantics they must handle in deployment.

To address the dual bottlenecks of scalable environment construction and trustworthy state verification, we present **OpenComputer**, a verifier-grounded framework for synthesizing verifiable software worlds within real desktop applications. Rather than treating verification as a downstream evaluation detail, OpenComputer makes verification the organizing principle of environment and task construction. It consists of four tightly coupled components as illustrated in Figure 1. First, it builds app-specific state verifiers that undergo a strict debug-fix-retry testing loop to reliably inspect software state through stable interfaces, defining exactly which task outcomes can be checked programmatically. Second, it further improves these verifiers through an execution-grounded self-evolution loop: calibration tasks are executed in sandboxed desktops, programmatic verifier outputs are compared against criterion-level LLM judgments, and verifier-side failures are used to refine checker logic, endpoints, or documentation. Third, on top of this verifier stack, OpenComputer synthesizes realistic user tasks through a structured pipeline that filters for difficulty, data generatability, and state inspectability. Finally, OpenComputer provides an evaluation harness that runs agents in fresh desktop sandboxes, records full screenshot-action trajectories, and scores each run by executing verifier commands over the resulting software state.

Empirically, OpenComputer remains challenging even for recent frontier computer-use agents. The strongest models such as GPT-5.6-Sol, Claude-Sonnet-5 and Kimi-K3 achieve task success rates in the mid-to-high 70s, leaving substantial room for improvement in reliable end-to-end task completion. Open-source agents lag substantially behind, with especially large drops relative to their reported performance on existing desktop benchmarks such as OSWorld (Xie et al., 2024). Our analysis further highlights the importance of verifier-grounded benchmark construction. Hard-coded verifiers align more closely with human adjudication than an agentic LLM judge, particularly when success depends on fine-grained application state that cannot be reliably inferred from screenshots alone.

We summarize our contributions as follows:

1. We introduce OpenComputer, a verifier-grounded framework for synthesizing realistic software worlds for computer-use agents, where the task descriptions, environments, and verifiers for evaluation are all automatically generated without relying on manual construction.

2. We empirically validate the reliability of this construction pipeline, showing that verifier-grounded evaluation aligns more closely with human adjudication than LLM-as-judge evaluation, and that the self-evolving verification layer can identify and repair verifier-side failures.

3. We instantiate a large-scale benchmark spanning 33 desktop applications and 1,000 finalized tasks, and evaluate frontier and open-source computer-use agents to show that realistic, verifier-grounded desktop workflows remain challenging for current systems.

2 Related Work

Benchmarks for Computer-Use Agents. Prior benchmarks for computer-use agents fall into two main categories: static trajectory datasets and interactive task environments. Static datasets such as Mind2Web (Deng et al., 2023) and Android in the Wild (Rawles et al., 2023) provide broad coverage of web or mobile interfaces through human demonstrations, but primarily evaluate offline action prediction. Interactive benchmarks more directly evaluate agents through environment feedback, including OSWorld (Xie et al., 2024) and Windows Agent Arena (Bonatti et al., 2024) for desktop operating-system tasks, BEARCUBS (Song et al., 2025b), RealWebAssist (Ye et al., 2026) for web tasks, WebArena (Zhou et al., 2023) and VisualWebArena (Koh et al., 2024) for realistic web navigation, WorkArena (Drouin et al., 2024) and Scuba (Dai et al., 2025) for enterprise web workflows, and AndroidWorld (Rawles et al., 2024) for mobile control. However, these benchmarks are still largely human-curated and often limited by the number of task instances, application domains, or manually written reward checks. In contrast, OpenComputer focuses on scaling computer-use environment construction itself.

Synthetic Environments for Agents. Recent work increasingly treats environment construction as a key bottleneck for training interactive agents. In tool-use and function-calling settings, AgentScaler builds simulated, database-backed API environments (Fang et al., 2025), Agent World Model scales code-driven multi-turn environments for RL (Wang et al., 2026), and Simia uses reasoning models to simulate environment feedback (Li et al., 2025b). These systems demonstrate the value of scalable interactive worlds, but primarily target abstract APIs or model-simulated feedback rather than native desktop software. Concurrent work synthesizes GUI and computer-use environments: InfiniteWeb builds functional websites with task-centric tests (Zhang et al., 2026), GUI-Genesis reconstructs mobile apps into lightweight web environments with code-native rewards (Cao et al., 2026), Gym-Anything (Aggarwal et al., 2026) uses an agentic creation-and-audit loop across software applications, and TermiGen (Zhu et al., 2026) and Scale-SWE (Zhao et al., 2026) automate executable environments for terminal and software-engineering agents. OpenComputer differs by making synthesis reward-aware from the outset: each generated desktop task is paired with verifiable reward implemented as executable checkers over inspectable application state, rather than relying on visual LLM judgments.

3 OpenComputer

We build OpenComputer as a verifier-grounded framework for constructing verifiable computer-use tasks in real desktop software environments. In this section, we first define the problem setup and then describe the four key layers of OpenComputer.

3.1 Problem Setup

Let $a \in \mathcal{A}$ denote a desktop application drawn from an application set $\mathcal{A}$, and let $g \in \mathcal{G}$ denote a natural-language user goal. Our objective is to synthesize a verifiable computer-use task instance

$$\tau = (x, e, c)$$

where x is the task description shown to the agent, e is an executable environment initialization procedure, and c is a set of machine-checkable success criteria. Each task is executed in a initial desktop sandbox state $s_0 \sim e$, and an agent interacts with the sandbox through screenshots and GUI actions to produce a final state s_T.

The core challenge is that realistic computer-use tasks require both environment construction and reliable verification. A goal g is only useful for benchmarking if we can: (1) materialize a coherent software world in which the task can be performed, and (2) determine from the resulting application state whether the goal has actually been achieved. We therefore cast environment construction as a constrained synthesis problem: given an application a and a goal g, generate a task instance τ

such that the initial environment is realistic, the target state is reachable through ordinary desktop interaction, and success can be checked programmatically.

OpenComputer solves this problem through three coupled components. First, a verifier generator

$$\mathcal{V}(a) \to V_a$$

builds an app-specific verifier V_a that exposes structured inspection and checking endpoints over the application state. Second, to repair residual verifier errors, a verifier-evolution procedure

$$\mathcal{U}(V_a, D_a) \to V_a^+$$

iteratively refines the verifier using calibration executions D_a collected from real agent runs. Third, a verifier-aware task and environment synthesis pipeline uses the resulting verifier stack to construct task instances: given an application a and a user goal g, it generates an executable environment initialization procedure

$$\mathcal{E}(a, g, V_a^+) \to e$$

together with a user-facing instruction x and machine-checkable success criteria c.

The final task synthesis pipeline combines these components to produce benchmark instances whose environments are executable and whose rewards are grounded in inspectable software state. The remainder of this section follows the same order: we describe how we build app-specific verifiers, how we evolve them from execution feedback, how we generate verifier-grounded task environments, and how we evaluate agents with structured reward computation.

3.2 Verification Stack

Verification is central to OpenComputer because realistic desktop tasks are only useful for training or evaluation when their outcomes can be checked reliably. Many success conditions are hidden in application state rather than visible in screenshots. The verification stack therefore defines what can be trusted as reward, and ensures that task generation and evaluation are grounded in reproducible, machine-checkable evidence.

3.2.1 Verifier Generation

Each supported application in the environment is paired with a synthetic Python verifier module that runs inside the sandbox and exposes a set of CLI subcommands with JSON outputs. These verifiers serve as stable inspection interfaces for downstream task generation and evaluation. Rather than focusing only on an application's primary document content, they are designed to cover all reliably inspectable state surfaces available for that application, including content state, preferences, plugins, history, bookmarks, file I/O, project structure, media state, graphical attributes, and metadata. In the notation of Section 3, for each application $a \in \mathcal{A}$ we instantiate an app-specific verifier $V_a = \mathcal{V}(a)$.

Inspection channels. To achieve this coverage, verifier endpoints query the most reliable application-specific inspection channels available in the sandbox. Depending on the target application, these channels may include browser debugging protocols, D-Bus, LibreOffice UNO, SQLite-backed profile databases, accessibility state, or direct parsing of saved files as shown in Figure 2. In this way, verification is grounded in the actual observable state of the application rather than in heuristic matching or surface-level script checks.

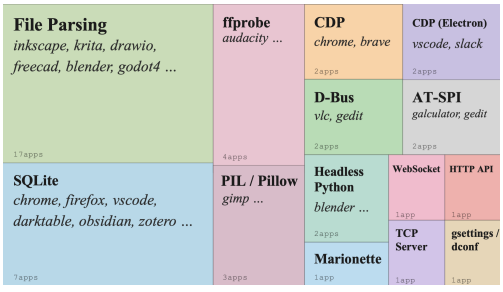


Figure 2: Example application endpoint specification used by OpenComputer verifiers.

Endpoint construction. Verifier development follows a fixed pipeline. The agent first enumerate the inspectable state surfaces of the target application and map each surface to a concrete verification channel. For example, browser-oriented tasks can often be verified through remote debugging APIs, office tasks through UNO interfaces or document parsing, and configuration-oriented tasks through SQLite databases. Based on this mapping, the agent implement query endpoints and `check-*` endpoints that expose these states as structured JSON, and then document them in an application-specific README so that later pipeline stages can treat the verifier as a well-defined interface.

Verifier testing protocol. The agent treat verifiers as software artifacts rather than ad hoc scripts. Each verifier includes an endpoint reference, a written test plan, and live integration tests against the real sandboxed application. The test plan covers expected assertions, realistic fixtures, positive and negative cases, JSON-validity checks, and common failure modes such as missing arguments, nonexistent paths, or inactive applications. For document-centric applications, the agent generate rich synthetic artifacts with realistic structure rather than toy files. Failed endpoints enter a debug-fix-retry loop until they become reliable, since unstable verifiers can produce misleading rewards.

3.2.2 SELF-EVOLVING VERIFICATION LAYER

After the initial verifier for an application is generated and passes its unit and integration tests, we further refine it through an self-evolving verification layer. The goal of this layer is to expose residual verifier issues that may not appear in synthetic tests alone, such as brittle assumptions about application schemas, incomplete endpoint coverage, or mismatches between documented and actual software behavior.

Calibration executions. For each application, we generate a small calibration set of approximately 15 easy-to-medium tasks that are expected to be solvable by a state-of-the-art computer-use agent. These tasks are not used to benchmark agent performance. Instead, they serve as execution-grounded probes for stress-testing the verifier before it is used for large-scale task synthesis and evaluation. We run the selected agent in a persistent desktop sandbox, record the full trajectory, and cache the resulting final environment state. The resulting execution can be viewed as taking the sandbox from an initialized state $s_0 \sim e$ to a realized terminal state s_T, and this recorded run is then treated as fixed throughout the refinement procedure.

Disagreement diagnosis. Given each fixed execution, an LLM evaluator inspects the trajectory, post-action observations, and available final-state evidence to produce a criterion-level diagnostic judgment. This judgment is used only to surface potential inconsistencies with the programmatic verifier, rather than as a ground-truth label. The verifier is independently executed against the same cached final state, and any disagreement between the two outputs is passed to a separate attribution agent. The attribution agent is an agentic LLM with direct access to the frozen sandbox state. For each disagreement, it receives the task specification, the LLM diagnosis, the verifier implementation and output, and the cached final environment. Unlike the initial evaluator, it can actively investigate the disagreement by inspecting application artifacts, files, databases, metadata, and other state exposed by the sandbox, as well as by issuing additional state queries and executing diagnostic commands. Based on this evidence, the agent determines whether the discrepancy is caused by an incorrect LLM judgment or by a concrete defect in the programmatic verifier, and records the supporting state evidence. Only disagreements for which the attribution agent identifies a specific verifier-side defect are eligible for verifier refinement.

Bounded verifier refinement. The verifier evolution step is restricted to the verification stack: it may modify checker code, endpoint implementations, or verifier documentation, but does not alter the cached trajectory, sandbox state, task objective, or expected outcome. A proposed repair is accepted only when the attribution stage identifies a concrete verifier-side failure, the revised verifier is consistent with directly inspectable application state on the frozen execution, and the updated implementation passes the corresponding unit and integration checks. When verifier-side issues are repaired, OpenComputer records the failed assumption and corrective action as an app-specific memory that can be reused during future verifier extension and task generation.

This layer provides an additional feedback channel between real software execution and verifier construction. By running strong agents on simple and moderate calibration tasks, OpenComputer can identify which endpoints are underspecified, and which verifier assumptions fail under realistic interaction. A concrete example of this stage is shown in Appendix A.

3.3 TASK GENERATION PIPELINE

Tasks are generated through a verifier-aware synthesis process that balances realism, difficulty, and checkability. The generator first proposes candidate tasks from the perspective of realistic user goals, without directly conditioning on the available verifier endpoints. This encourages task diversity and

Table 1: Summary statistics of the OpenComputer benchmark.

Applications	Tasks	Avg. Verifier Endpoints / App	Avg. Checks / Task	Avg. Seed Files / Task
33	1000	17.7	6.9	1.3

avoids overfitting the benchmark to what is already easy to check. Candidate tasks are then filtered for complexity and data generatability: we prioritize multi-step workflows in the upper half of the difficulty scale and reject tasks that are too short, overly linear, trivial, or difficult to instantiate with coherent input artifacts.

Accepted proposals are then grounded in the verification stack. If the intended state can be checked by an existing endpoint, the task is retained directly. If the outcome is inspectable but not yet exposed, the verifier is extended with a new endpoint following the verifier-generation procedure in Section 3.2.1. Finally, the system materializes each task by generating and packaging the required files, folders, profiles, configurations, or other input artifacts. Each finalized task is stored as a `task.json` instance $\tau = (x, e, c)$, where x is the user-facing instruction, e initializes the sandbox, and c specifies the executable success criteria. This process turns open-ended desktop workflows into reproducible benchmark instances with machine-checkable rewards.

To prevent coverage collapse, the task generator includes a task-extension workflow. We periodically review each application's task set by feature area, identify missing or repetitive workflows, and prioritize gaps with reliable verification paths. New candidate tasks for these gaps are then passed through the same four-stage proposal, filtering, verification, and environment-synthesis pipeline.

3.4 Evaluation Harness and Reward Computation

At evaluation time, the harness uploads the verifier and task artifacts into a fresh sandbox, launches the target application, and runs a screenshot-action loop with the chosen agent. At each step, the system captures the current desktop framebuffer, feeds it to the agent, executes the predicted action, and logs the resulting reasoning, action sequence, and screenshot. In the formalization above, evaluation executes the task instance $\tau = (x, e, c)$ by first sampling $s_0 \sim e$ and then checking whether the agent's interaction trajectory reaches a terminal state s_T that satisfies c.

After the agent stops or reaches a step budget, the harness attempts a final save action for applications where persistence matters. Verification is then performed by executing the task's checker commands inside the sandbox. The task reward is the fraction of checks that pass, $R = N_{\text{pass}}/N_{\text{total}}$. This scoring scheme supports partial credit while preserving exact, machine-checkable success conditions.

3.5 OpenComputer Release

We release OpenComputer as an extensible infrastructure for both training and evaluating computer-use agents in verifiable software environments. The release includes 33 desktop applications and 1,000 finalized tasks, together with app-specific verifier modules, task specifications, environment-initialization scripts, and an execution harness. Summary statistics of the released synthetic benchmark are reported in Table 1. OpenComputer supports both local and cloud-scale execution. Users can run tasks locally with Docker-based sandboxes, deploy the same stack on self-hosted or cloud machines such as AWS, Tencent Cloud, or E2B for parallel rollouts. Beyond fixed evaluation, OpenComputer also supports training pipelines: researchers can collect trajectories, filter successful or partially successful runs, build SFT data, and use machine-checkable rewards for RL or rejection sampling. Users can also extend existing applications or add new ones through the same verifier-guided task and environment synthesis workflow.

4 Experiment

4.1 Experimental Setup

We design our experiments to evaluate whether OpenComputer provides reliable and challenging software-world environments for computer-use agents. Our evaluation focuses on two questions: (1)

Table 2: Performance and efficiency comparison across computer-use agents on our benchmark, with OSWorld-Verified reported as an external reference when available. The OSWorld column summarizes the publicly reported OSWorld-Verified score for the corresponding model. Success rate reports the fraction of tasks completed successfully. Average steps and time (seconds) per step capture interaction efficiency. Average reward measures the mean checklist-based score over all tasks.

Model	OSWorld	Success Rate	Avg. Steps	Time/Step	Avg. Reward
Claude-Sonnet-5	81.2%	77.2%	32.0	16.1 s	87.3%
GPT-5.6-Sol	83.0%	76.5%	11.5	14.7 s	89.8%
Kimi-K3	84.8%	75.8%	19.7	24.7 s	85.4%
GPT-5.4	75.0%	68.3%	19.0	16.5 s	84.2%
Claude-Sonnet-4.6	72.5%	64.4%	31.5	20.8 s	76.6%
GPT-5.6-Terra	–	63.3%	10.2	6.7 s	83.9%
Gemini-3.7-Flash	–	36.7%	16.9	10.3 s	54.2%
Qwen-3.5-27B	56.2%	32.3%	33.1	57.3 s	59.4%
EvoCUA-8B	46.1%	10.9%	67.0	9.7 s	38.1%
Qwen-3.5-9B	41.8%	7.8%	39.3	17.8 s	31.7%
GUI-OWL-1.5-8B	52.3%	5.7%	73.6	9.43 s	27.8%

whether current frontier and open-source agents can complete the synthesized tasks, and (2) whether our verifier-grounded reward computation can measure both exact success and partial progress across heterogeneous desktop applications.

Benchmark. We evaluate agents on the finalized OpenComputer task suite. Each task consists of a natural-language instruction, an executable sandbox initialization, and a set of machine-checkable success criteria. The benchmark spans 33 desktop applications. For each task, the agent is placed in a fresh desktop sandbox initialized with the required files, profiles, configuration state, and application artifacts. The agent then interacts with the live GUI through screenshots and desktop actions until it stops or reaches the step budget. We include OSWorld-Verified (Xie et al., 2024) as an external reference to contextualize model performance against a widely used desktop-agent benchmark.

Models. We evaluate a mixture of frontier proprietary agents and open-source computer-use models. The main models include GPT-5.6-Sol (OpenAI, 2026b), GPT-5.6-Terra (OpenAI, 2026b), GPT-5.4 (OpenAI, 2026a), Claude-Sonnet-5 (Anthropic, 2026b), Claude-Sonnet-4.6 (Anthropic, 2026a), Kimi-K3 (Kimi Team, 2026), Gemini-3.7-Flash (Google, 2026), Qwen-3.5-27B (Qwen Team, 2026), Qwen-3.5-9B (Qwen Team, 2026), EvoCUA-8B (Xue et al., 2026), and GUI-OWL-1.5-8B (Xu et al., 2026). All open-source models except Kimi-K3 (which we use the official APIs) are deployed with two H100 GPUs.

Metrics. We report both task-level and criterion-level metrics. The primary task-level metric is *success rate*, defined as the fraction of tasks for which all required criteria are satisfied. Because many desktop tasks contain multiple independent requirements, we also report *average reward*, defined as the mean fraction of passed verifier checks, this metric gives partial credit when an agent completes some but not all required subtasks. To measure efficiency, we additionally report the average number of interaction steps and the average wall-clock time per step.

4.2 Main Results Analysis

Table 2 reports the overall performance and efficiency of representative frontier and open-source computer-use agents on OpenComputer. The latest frontier agents substantially improve over earlier generations, but the benchmark remains unsaturated. Claude-Sonnet-5 achieves the highest observed task success rate at 77.2%, followed closely by GPT-5.6-Sol at 76.5% and Kimi-K3 at 75.8%. GPT-5.6-Sol obtains the highest average reward of 89.8%, compared with 87.3% for Claude-Sonnet-5 and 85.4% for Kimi-K3. This indicates that frontier models can often make substantial partial progress, but reliable end-to-end task completion in realistic desktop software remains unsaturated.

Efficiency varies substantially across frontier agents, both in interaction overhead and token consumption. GPT-5.6-Terra requires only 10.2 interaction rounds on average and achieves the lowest

per-step latency, while GPT-5.6-Sol reaches 76.5% success in just 11.5 rounds. By comparison, Claude-Sonnet-5 requires 32.0 interaction rounds despite achieving a similar success rate of 77.2%. This difference is partly explained by action granularity: GPT-5.6 executes approximately three computer-control actions per model step on average, allowing multiple low-level operations to be completed within a single model–environment interaction. Token usage also differs markedly across models. Kimi-K3 is particularly token-intensive, averaging 43.9K input tokens and 652 output tokens per model call, compared with 25.4K input tokens and 167 output tokens for GPT-5.6-Sol.

The external OSWorld-Verified scores provide additional context. Several open-source models have moderate reported OSWorld performance, but their success rates drop substantially on OpenComputer. For example, GUI-OWL-1.5-8B has a reported OSWorld score of 52.3%, but achieves only 5.7% success on our benchmark; EvoCUA-8B similarly drops from 46.1% on OSWorld to 10.9% on OpenComputer. This gap suggests that these models have limited cross-benchmark generalization, and that strong performance on existing desktop benchmarks does not necessarily transfer to the broader and more heterogeneous software settings covered by OpenComputer.

5 Analysis

5.1 Agentic LLM-as-Judge vs. Hard-Coded Verification

We use LLM-as-judge as a two-stage agentic pipeline. The judge first reads the reasoning and action trace to identify a small set of steps that are most likely to contain evidence for each criterion. It then scores each criterion from these step's corresponding screenshots, with the option to retrieve more screenshots when existing ones are not sufficient. To quantify the gap between these two evaluation strategies, we sample 120 tasks and send the same completed trajectories to human annotators. We then score the 120 trajectories with two automated evaluators: an LLM judge and our final hard-coded verifier. We use the same per-item checklist for both methods. For each task, the item-level decisions are aggregated into a task-level verdict, and we compare that verdict against the human label.

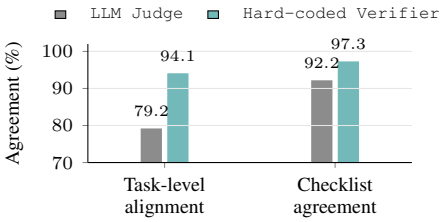


Figure 3: Alignment with human adjudication on a 120-task held-out audit set excluded from verifier evolution.

Figure 3 shows that the hard-coded verifier aligns much better with human judgment at both levels: it matches human verdicts on 113 out of 120 tasks, whereas the LLM judge reaches 95 out of 120, and it also achieves higher per-item checklist agreement with human annotations (97.3% versus 92.2%). In dense desktop interfaces, semantically important mistakes are often visually tiny: a model may type two tokens into one spreadsheet cell instead of two adjacent cells or edit a field inside a collapsed panel that is only partially visible. These runs can look approximately correct from pixels alone. A hard-coded verifier instead reads the exact state of the application and can therefore distinguish near-miss visual outputs from true task completion.

Table 3: Overall GUI–CLI pass-rate and execution-time (per task) comparison. For the CLI Agent, we use Claude Code (v2.1.129).

Setting	Model	Success (%)	Time (s)
GUI	GPT-5.4	75.2	288
GUI	Claude Sonnet 4.6	72.9	622
CLI	Claude Sonnet 4.6	67.1	141

The gap is even larger for applications with heavy terminal usage or agent with mixed action spaces. In environments such as Blender or developer tools, success often depends on scrollback logs or intermediate artifacts that are not simultaneously visible on screen. Appendix B presents two concrete visual examples of these failure modes.

5.2 Comparing GUI Agents with CLI Agents

Because OpenComputer verifies final application state rather than a particular interaction trace, it can in principle evaluate agents that reach the same target state through different control interfaces. We therefore compare GUI and CLI agents on a shared CLI-compatible subset to test whether

OpenComputer 's verifier-grounded tasks transfer beyond screenshot-action GUI control, and to quantify the trade-off between visual grounding and programmatic execution efficiency. Since many OpenComputer applications are inherently GUI-centric and cannot be meaningfully executed from a terminal alone, we construct a controlled subset by removing applications whose tasks are not suitable for CLI execution. This leaves 14 applications and 343 tasks that are compatible with both GUI and CLI settings. For the CLI setting, we use Claude Sonnet 4.6 with Claude Code, where the agent can combine CLI-Anything (HKUDS, 2026) skills, Bash commands, and Python scripts to inspect files, manipulate artifacts, and execute application-specific operations.

Table 3 shows the comparison. On this shared subset, GUI agents still achieve higher pass rates than the CLI agent. This suggests that even when tasks are selected to be CLI-solvable, visual interaction provides useful grounding for many desktop workflows. At the same time, the CLI agent is substantially faster. Claude Code completes tasks in 141 seconds on average, compared with 288 seconds for GPT-5.4 GUI control and 622 seconds for Claude Sonnet 4.6 GUI control. This reflects the efficiency advantage of command-line execution: the agent can bypass slow screenshot-action interaction loops and directly manipulate files, run scripts, or invoke application-level tools.

5.3 Ablation: Self-Evolving Verification

We ablate the contribution of the self-evolving verification layer by measuring how often it can identify and repair checker-side errors. We generate 450 simple calibration tasks and run the self-evolution procedure with a maximum repair budget of three iterations per task. These tasks are used only to probe verifier reliability, not to measure agent capability.

Among the 450 calibration executions, 159 tasks exhibit at least one disagreement between the programmatic checker and the reference evaluation. After categorizing the disagreement source, we find that 76 cases are attributable to checker-side errors rather than agent failures. The self-evolution procedure repairs 68 of these 76 checker-side cases, corresponding to an 89.4% repair rate.

Table 4: Repair efficiency and human-checker agreement improvement from self-evolving verification.

Metric	**Value**
Fixed in 1 round	47
Fixed in 2 rounds	15
Fixed in 3 rounds	6
Not fixed within budget	8
Agreement before evolution	85.0%
Agreement after evolution	94.1% (+9.1%)

Table 4 further breaks down the repair process. Most checker-side errors are fixed quickly: 47 cases are repaired after one iteration, 15 after two iterations, and only 6 require the full three-iteration budget. The remaining 8 cases are not resolved within the budget. We compare the pre- and post-evolution checkers on the same 120-task human-annotated comparison set used in Section 5.1. As a result, human-checker agreement improves from 85.0% before self-evolution to 94.1% after self-evolution. This suggests that the self-evolving layer provides a useful debugging signal for improving verifier reliability while preserving programmatic, auditable evaluation.

6 Conclusion

We introduced **OpenComputer**, a verifier-grounded framework for building verifiable software worlds for computer-use agents. OpenComputer makes inspectable application state a first-class design constraint throughout verifier construction, task synthesis, and benchmark execution. This design enables the automatic generation of executable desktop tasks with machine-checkable success criteria, while preserving the diversity and realism of real software workflows. In its current form, OpenComputer covers 33 desktop applications and 1,000 finalized tasks, spanning browsers, office tools, creative software, development environments, communication tools, and system utilities. More broadly, we view OpenComputer as infrastructure for scaling computer-use research in a principled way. Progress in this area depends not only on stronger models, but also on trustworthy environments, grounded rewards, reproducible task construction pipelines, and verifiers that can support both evaluation and training. We hope this work, together with the released repository, can serve as a foundation for future research by providing a scalable environment for studying agent reliability, collecting grounded trajectories, and improving computer-use agents with auditable feedback.

AI USE STATEMENT

Generative AI tools were used solely for limited assistance with language polishing, formatting, routine coding, and debugging. The research, analysis, interpretation of results, and preparation of the final content were carried out by the human authors. All results and citations were independently verified by the authors, who take full responsibility for the work.

REFERENCES

Saaket Agashe, Jiuzhou Han, Shuyu Gan, Jiachen Yang, Ang Li, and Xin Eric Wang. Agent s: An open agentic framework that uses computers like a human. In *The Thirteenth International Conference on Learning Representations*.

Saaket Agashe, Kyle Wong, Vincent Tu, Jiachen Yang, Ang Li, and Xin Eric Wang. Agent s2: A compositional generalist-specialist framework for computer use agents. *arXiv preprint arXiv:2504.00906*, 2025.

Pranjal Aggarwal, Graham Neubig, and Sean Welleck. Gym-anything: Turn any software into an agent environment. *arXiv preprint arXiv:2604.06126*, 2026.

Anthropic. Introducing claude sonnet 4.6. `https://www.anthropic.com/news/claude-sonnet-4-6`, 2026a.

Anthropic. Introducing claude sonnet 5, 2026b. URL `https://www.anthropic.com/news/claude-sonnet-5`. Published June 30, 2026.

Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, et al. Windows agent arena: Evaluating multi-modal os agents at scale. *arXiv preprint arXiv:2409.08264*, 2024.

Yuan Cao, Dezhi Ran, Mengzhou Wu, Yuzhe Guo, Xin Chen, Ang Li, Gang Cao, Gong Zhi, Hao Yu, Linyi Li, et al. Gui-genesis: Automated synthesis of efficient environments with verifiable rewards for gui agent post-training. *arXiv preprint arXiv:2602.14093*, 2026.

Chaoqun Cui, Jing Huang, Shijing Wang, Liming Zheng, Qingchao Kong, and Zhixiong Zeng. Agentic reward modeling: Verifying gui agent via online proactive interaction. *arXiv preprint arXiv:2602.00575*, 2026.

Yutong Dai, Krithika Ramakrishnan, Jing Gu, Matthew Fernandez, Yanqi Luo, Viraj Prabhu, Zhenyu Hu, Silvio Savarese, Caiming Xiong, Zeyuan Chen, et al. Scuba: Salesforce computer use benchmark. *arXiv preprint arXiv:2509.26506*, 2025.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114, 2023.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H Laradji, Manuel Del Verme, Tom Marty, Léo Boisvert, Megh Thakkar, Quentin Cappart, David Vazquez, et al. Workarena: How capable are web agents at solving common knowledge work tasks? *arXiv preprint arXiv:2403.07718*, 2024.

Runnan Fang, Shihao Cai, Baixuan Li, Jialong Wu, Guangyu Li, Wenbiao Yin, Xinyu Wang, Xiaobin Wang, Liangcai Su, Zhen Zhang, et al. Towards general agentic intelligence via environment scaling. *arXiv preprint arXiv:2509.13311*, 2025.

Google. Gemini 3.7 Flash: Our Most Intelligent Workhorse Model, 2026. URL `https://blog.google/innovation-and-ai/models-and-research/gemini-models/introducing-gemini-3-7-flash/`. Published August 13, 2026.

Yanheng He, Jiahe Jin, Shijie Xia, Jiadi Su, Runze Fan, Haoyang Zou, Xiangkun Hu, and Pengfei Liu. Pc agent: While you sleep, ai works–a cognitive journey into digital world. *arXiv preprint arXiv:2412.17589*, 2024.

HKUDS. CLI-Anything: Making all software agent-native. https://github.com/HKUDS/CLI-Anything, 2026. GitHub repository. Accessed: 2026-05-02.

Seungone Kim, Jamin Shin, Yejin Cho, Joel Jang, Shayne Longpre, Hwaran Lee, Sangdoo Yun, Seongjin Shin, Sungdong Kim, James Thorne, et al. Prometheus: Inducing fine-grained evaluation capability in language models. In *The Twelfth International Conference on Learning Representations*, 2023.

Kimi Team. Kimi K3: Open Frontier Intelligence. *arXiv preprint arXiv:2607.24653*, 2026. URL https://arxiv.org/abs/2607.24653.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 881–905, 2024.

Dawei Li, Bohan Jiang, Liangjie Huang, Alimohammad Beigi, Chengshuai Zhao, Zhen Tan, Amrita Bhattacharjee, Yuxuan Jiang, Canyu Chen, Tianhao Wu, et al. From generation to judgment: Opportunities and challenges of llm-as-a-judge. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 2757–2791, 2025a.

Yuetai Li, Huseyin A Inan, Xiang Yue, Wei-Ning Chen, Lukas Wutschitz, Janardhan Kulkarni, Radha Poovendran, Robert Sim, and Saravan Rajmohan. Simulating environments with reasoning models for agent training. *arXiv preprint arXiv:2511.01824*, 2025b.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-eval: Nlg evaluation using gpt-4 with better human alignment. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 2511–2522, 2023.

Dang Nguyen, Jian Chen, Yu Wang, Gang Wu, Namyong Park, Zhengmian Hu, Hanjia Lyu, Junda Wu, Ryan Aponte, Yu Xia, et al. Gui agents: A survey. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 22522–22538, 2025.

OpenAI. Gpt-5.4 model. https://developers.openai.com/api/docs/models/gpt-5.4, 2026a.

OpenAI. GPT-5.6 System Card, 2026b. URL https://deploymentsafety.openai.com/gpt-5-6. Published July 9, 2026.

Qwen Team. Qwen3.5 model family. https://huggingface.co/collections/Qwen/qwen35, 2026.

Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. Androidinthewild: A large-scale dataset for android device control. *Advances in Neural Information Processing Systems*, 36:59708–59728, 2023.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, et al. Androidworld: A dynamic benchmarking environment for autonomous agents. *arXiv preprint arXiv:2405.14573*, 2024.

Linxin Song, Yutong Dai, Viraj Prabhu, Jieyu Zhang, Taiwei Shi, Li Li, Junnan Li, Silvio Savarese, Zeyuan Chen, Jieyu Zhao, et al. Coact-1: Computer-using agents with coding as actions. *arXiv preprint arXiv:2508.03923*, 2025a.

Yixiao Song, Katherine Thai, Chau Minh Pham, Yapei Chang, Mazin Nadaf, and Mohit Iyyer. Bearcubs: A benchmark for computer-using web agents. *arXiv preprint arXiv:2503.07919*, 2025b.

Marta Sumyk and Oleksandr Kosovan. Cuaaudit: Meta-evaluation of vision-language models as auditors of autonomous computer-use agents. *arXiv preprint arXiv:2603.10577*, 2026.

Aman Singh Thakur, Kartik Choudhary, Venkat Srinik Ramayapally, Sankaran Vaidyanathan, and Dieuwke Hupkes. Judging the judges: Evaluating alignment and vulnerabilities in llms-as-judges. In *Proceedings of the Fourth Workshop on Generation, Evaluation and Metrics (GEM^2)*, pp. 404–430, 2025.

Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, Binghuai Lin, Yunbo Cao, Lingpeng Kong, Qi Liu, Tianyu Liu, et al. Large language models are not fair evaluators. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9440–9450, 2024.

Zhaoyang Wang, Canwen Xu, Boyi Liu, Yite Wang, Siwei Han, Zhewei Yao, Huaxiu Yao, and Yuxiong He. Agent world model: Infinity synthetic environments for agentic reinforcement learning. *arXiv preprint arXiv:2602.10090*, 2026.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

Haiyang Xu, Xi Zhang, Haowei Liu, Junyang Wang, Zhaozai Zhu, Shengjie Zhou, Xuhao Hu, Feiyu Gao, Junjie Cao, Zihua Wang, et al. Mobile-agent-v3. 5: Multi-platform fundamental gui agents. *arXiv preprint arXiv:2602.16855*, 2026.

Yiheng Xu, Dunjie Lu, Zhennan Shen, Junli Wang, Zekun Wang, Yuchen Mao, Caiming Xiong, and Tao Yu. Agenttrek: Agent trajectory synthesis via guiding replay with web tutorials. *arXiv preprint arXiv:2412.09605*, 2024.

Taofeng Xue, Chong Peng, Mianqiu Huang, Linsen Guo, Tiancheng Han, Haozhe Wang, Jianing Wang, Xiaocheng Zhang, Xin Yang, Dengchang Zhao, et al. Evocua: Evolving computer use agents via learning from scalable synthetic experience. *arXiv preprint arXiv:2601.15876*, 2026.

Suyu Ye, Haojun Shi, Darren Shih, Hyokun Yun, Tanya G Roosta, and Tianmin Shu. Realwebassist: A benchmark for long-horizon web assistance with real-world users. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 34441–34449, 2026.

Ziyun Zhang, Zezhou Wang, Xiaoyi Zhang, Zongyu Guo, Jiahao Li, Bin Li, and Yan Lu. Infiniteweb: Scalable web environment synthesis for gui agent training. *arXiv preprint arXiv:2601.04126*, 2026.

Jiale Zhao, Guoxin Chen, Fanzhe Meng, Minghao Li, Jie Chen, Hui Xu, Yongshuai Sun, Wayne Xin Zhao, Ruihua Song, Yuan Zhang, et al. Immersion in the github universe: Scaling coding agents to mastery. *arXiv preprint arXiv:2602.09892*, 2026.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

Kaijie Zhu, Yuzhou Nie, Yijiang Li, Yiming Huang, Jialian Wu, Jiang Liu, Ximeng Sun, Zhenfei Yin, Lun Wang, Zicheng Liu, et al. Termigen: High-fidelity environment and robust trajectory synthesis for terminal agents. *arXiv preprint arXiv:2602.07274*, 2026.

A CASE STUDY: SELF-EVOLVING VERIFICATION IN A PROGRAMMATIC VERIFIER

This appendix provides a concrete example of the optional self-evolving verification layer described in Section 3.2.2. The goal of this layer is to use execution-grounded feedback to refine the verification stack and improve future task synthesis. The example below illustrates a common failure mode: the agent completed the task, but the verifier queried an outdated application schema and therefore incorrectly marked several satisfied criteria as failed. By comparing the programmatic verdict against an LLM reference judgment on a fixed trajectory, the system identifies the verifier-side defect and updates the verification logic accordingly.

Task. We use the darktable task `darktable_batch_rate_and_tag`. The agent is instructed to import three images, create the tag `batch_processed`, attach the tag to all three images, and assign ratings of one, three, and five stars to `img_001.png`, `img_002.png`, and `img_003.png`, respectively. The recorded run was produced by `kimi-k2.6` and completed in 53 interaction steps. The trajectory was then frozen and reused throughout the self-evolution procedure.

Reference judgment. An LLM judge inspected the full trajectory, post-action screenshots, and final state. It judged all ten criteria as satisfied: the three images were imported, the tag `batch_processed` was visible and attached to all images, and the final star ratings matched the requested values. In particular, the judge found that the tag was created while all three images were selected, and that the final state showed the expected image information and rating flags for each image.

Round	Source	Passed	Failed	Divergences
0	Programmatic verifier before evolution	6	4	4
0	LLM reference judgment	10	0	–
1	Programmatic verifier after evolution	10	0	0

Table 5: Self-evolution outcome for `darktable_batch_rate_and_tag`. The initial verifier incorrectly failed four tag-related criteria. After updating the verification logic using execution-grounded feedback, the programmatic verdict agreed with the LLM reference on all ten criteria.

Detected disagreement. The comparator found four disagreements between the LLM reference judgment and the programmatic verifier. All four involved tag state: whether the tag `batch_processed` existed, and whether it was attached to each of the three imported images. In each case, the LLM judge returned TRUE, while the verifier returned FALSE. The comparator classified all four disagreements as verifier-side failures, meaning that the task had been completed but the checker misjudged the final state.

Table 6: Criterion-level disagreements before self-evolution. All failures share the same root cause: the verifier checked tag metadata using an outdated database assumption.

Criterion	Description	Verifier	Judge	Classification
3	Tag `batch_processed` exists	False	True	`verifier_wrong`
4	`img_001` has tag	False	True	`verifier_wrong`
5	`img_002` has tag	False	True	`verifier_wrong`
6	`img_003` has tag	False	True	`verifier_wrong`

Root cause. All four failures were caused by a single verifier bug. The darktable verifier assumed that the table `tags` lived in `library.db`. In the current darktable state, however, tag definitions are stored in `data.db`, while image-tag associations remain in `library.db`. As a result, tag-related SQL queries failed with a missing-table error and were counted as negative verifier results, even though the final application state contained the expected tag assignments.

Verifier evolution. The self-evolving layer was allowed to modify only the verifier implementation and documentation, not the agent trajectory, sandbox state, task specification, or expected output. The verifier update made three changes. First, `check_tag_exists` and the corresponding tag-query endpoint were rerouted to query `data.db`. Second, the image-tag checker was rewritten to join `library.db`'s `tagged_images` table with `data.db`'s `tags` table. Third, the verifier documentation was updated to reflect the actual darktable schema. These changes preserve the same public checker interface while aligning the internal inspection logic with the real application state.

Table 7: Summary of the verifier evolution. The public checker interface was unchanged; only the internal SQL source and join path were updated.

Check	Before evolution	After evolution
Tag existence	Query `main.tags` inside `library.db`.	Query `main.tags` inside `data.db`.
Image-tag assignment	Join `main.tagged_images` with `main.tags` inside `library.db`.	Join `main.tagged_images` from `library.db` with `data.tags` from attached `data.db`.

Before and after. The tag-existence check required only changing the database queried by the existing SQL statement:

```
# Before
rows = _query_sqlite(LIBRARY_DB, sql, (tag_name, f"%|{tag_name}"))

# After
rows = _query_sqlite(DATA_DB, sql, (tag_name, f"%|{tag_name}"))
```

The image-tag checker required a cross-database join:

```
# Before
SELECT t.id AS tag_id, t.name AS tag_name
FROM main.tagged_images ti
JOIN main.tags t ON ti.tagid = t.id
WHERE ti.imgid = ? AND (t.name = ? OR t.name LIKE ?)

# After
SELECT t.id AS tag_id, t.name AS tag_name
FROM main.tagged_images ti
JOIN data.tags t ON ti.tagid = t.id
WHERE ti.imgid = ? AND (t.name = ? OR t.name LIKE ?)
```

Outcome. After self-evolution, the verifier was re-executed on the same cached final state. The updated verifier passed all ten criteria and had zero remaining divergences from the LLM reference judgment. This example demonstrates how the self-evolving verification layer provides an additional feedback channel for the synthesis pipeline: it identifies brittle verifier assumptions, such as application schema drift, updates the executable inspection logic, and records which application states require more careful grounding in future task generation. In this way, OpenComputer improves its verifier stack over time while preserving the core principle that agent performance is scored by executable, application-grounded checks.

B Case Study: Comparison between LLM as Judge and Hard-Coded Verifier

This appendix illustrates why we use LLM-as-judge only as a reference signal for verifier debugging, rather than as the final benchmark reward.

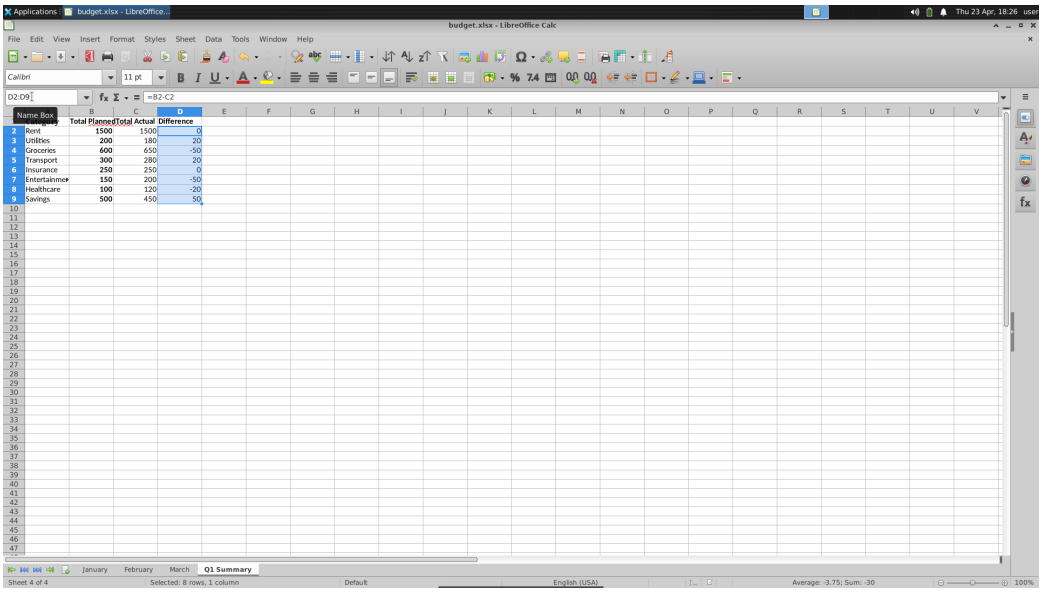


Figure 4: A dense spreadsheet-style interface where the visual output looks almost correct, but the state is wrong: the agent typed one long token into a single cell instead of filling two adjacent cells. This is difficult to judge reliably from pixels alone, but trivial to detect from the underlying workbook state.

Failure mode 1: dense interfaces hide exact state. In spreadsheet-like applications, the difference between success and failure may be encoded in a single cell boundary, a hidden formula, or a small formatting change. Figure 4 shows a representative example in which the agent types `alpha beta` into one cell, although the task requires `alpha` and `beta` to be entered into two adjacent cells. To a screenshot-based judge, the rendered sheet still looks broadly plausible, especially when grid lines are thin or the screenshot is downsampled. A hard-coded verifier can instead read the workbook state directly and determine exactly which cell contains which value.

Failure mode 2: terminal-heavy tasks exceed screenshot context. For terminal-centric or mixed GUI-and-terminal workloads, the problem is not just fine-grained visual ambiguity but limited observability. Figure 5 shows a case where the terminal contains the decisive evidence: an error line and a missing output artifact. A screenshot captures only one scroll position and one pane layout, so the judge must infer whether earlier logs, filesystem state, and intermediate outputs are consistent with task completion.

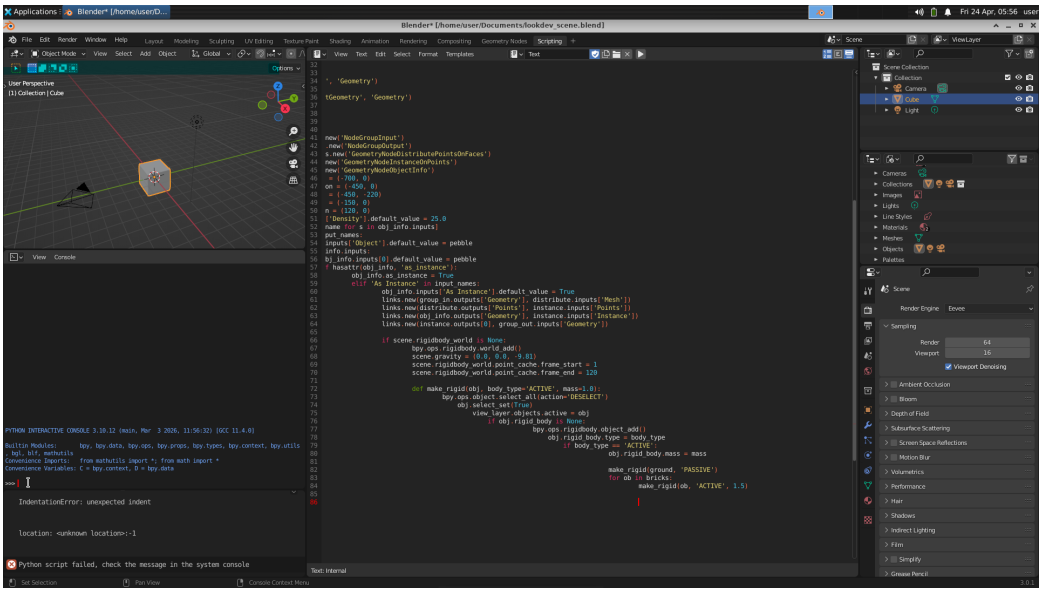


Figure 5: A terminal-heavy workflow where the decisive evidence lives in log lines, exit codes, and filesystem artifacts rather than in a clean final screenshot. Programmatic verifiers can inspect these sources directly, whereas screenshot-based judges only observe a narrow and potentially misleading window.

Implication for OpenComputer. These examples motivate the separation of roles in OpenComputer. We use LLM-as-judge as a flexible, high-level reference that helps detect verifier bugs, underspecified criteria, and other pipeline issues during task construction. But we reserve final scoring for hard-coded verifiers that inspect application-grounded state directly. This choice makes rewards reproducible, auditable, and sensitive to the exact success conditions that the benchmark is meant to evaluate.

C Examples of Generated Verifiable Tasks

This section shows representative tasks generated by OpenComputer across different desktop applications. Each task is paired with executable verification criteria that check the resulting application state, files, metadata, or persistent side effects.

Zotero: Three-Level Collection Hierarchy

Application. `zotero`

Task. Launch Zotero with the pre-seeded library at `/home/user/Zotero/zotero.sqlite`. Under `My Library`, create a top-level collection named `Papers`, a subcollection named `Vision`, and a sub-subcollection named `Object Detection`. Each collection must be a direct child of the previous level. Leave Zotero running so the updated library state is saved.

Initial files. `zotero.sqlite` at `/home/user/Zotero/zotero.sqlite`.

Representative verification checks.

- The collection `Papers` exists in the Zotero library.
- The collection `Vision` exists.
- The collection `Object Detection` exists.
- `Vision` is a direct child of `Papers`.
- `Object Detection` is a direct child of `Vision`.

MuseScore: Piano Solo to Piano Quartet

Application. `musescore3`

Task. Open the score `/home/user/Documents/piano_sketch.mscz`, which contains a single Piano part in C major, 4/4 time, and four measures of quarter notes. Arrange the solo as a piano quartet by adding `Violin`, `Viola`, and `Violoncello` parts below the Piano. Copy the Piano melody to the Violin staff. On the Viola staff, enter one whole note per measure with pitches `A3`, `D4`, `G3`, and `C4`. On the Violoncello staff, enter a walking bass using four quarter notes per measure: `C3`, `E3`, `G3`, `E3`; `F3`, `A3`, `C4`, `A3`; `G3`, `B3`, `D4`, `B3`; and `C3`, `E3`, `G3`, `C4`. Add dynamics `mp`, `mf`, `p`, and `f` to measures 1–4, respectively. Add staccato articulations to all four Violoncello notes in measure 4 and nowhere else. Save the score, then export it as uncompressed MusicXML to `/home/user/Documents/piano_sketch.musicxml`.

Initial files. `piano_sketch.mscz` at `/home/user/Documents/piano_sketch.mscz`.

Representative verification checks.

- The original `.mscz` file still exists at the expected path.
- The saved score has exactly four parts: `Piano`, `Violin`, `Viola`, and `Violoncello`.
- The score still has four measures and remains in 4/4 time.
- The score contains at least 52 notes, covering the original Piano melody, copied Violin melody, Viola whole notes, and Violoncello bass line.
- Measures 1–4 contain the expected dynamics: `mp`, `mf`, `p`, and `f`.
- Measure 4 contains staccato articulations.
- The total number of staccato articulations in the score is exactly 4.
- The exported MusicXML file exists at `/home/user/Documents/piano_sketch.musicxml`.
- The exported MusicXML file contains exactly four parts.

LibreOffice Calc: Sales Commission Spreadsheet

Application. `libreoffice_calc`

Task. Open `/home/user/Documents/commissions.xlsx`. In the `Sales` sheet, add bold headers `Commission Rate` and `Commission Amount`. Use a nested `IF` formula to assign commission rates based on monthly sales: 10% for sales above 20,000, 8% for sales at least 10,000, and 5% otherwise. Compute commission amounts for all 20 records, create a new `Commission Summary` sheet, and add formulas for total sales, total commission, and average commission rate. Save the workbook.

Initial files. `commissions.xlsx` at `/home/user/Documents/commissions.xlsx`.

Representative verification checks.

- Cells `D1` and `E1` contain the expected bold headers.
- Cell `D2` contains an `IF` formula and evaluates to the expected commission rate.
- Cell `E2` references `C2` and `D2` and computes the expected commission amount.
- The copied formulas produce expected values for low-, mid-, and high-tier sales examples.
- The `Commission Summary` sheet exists.
- The summary sheet contains the expected labels and formulas for total sales, total commission, and average commission rate.
- The workbook is saved.

Obsidian: Recipe Vault Construction

Application. `obsidian`

Task. Open the Obsidian vault at `/home/user/Documents/RecipeVault`. Create folders for `Italian`, `Asian`, and `Desserts`. Add four recipe notes with YAML frontmatter, headings, ingredients, instructions, tags, and internal links: `Carbonara.md`, `Cacio e Pepe.md`, `PadThai.md`, and `Tiramisu.md`. Finally, create a root `Index.md` note linking to all recipes and tagged as an index.

Initial files. None.

Representative verification checks.

- The `Italian`, `Asian`, and `Desserts` folders exist in the vault.
- Recipe notes contain the expected frontmatter fields, such as `cuisine`, `servings`, and `time`.
- `Carbonara` links to `Cacio e Pepe`, and `Cacio e Pepe` links back to `Carbonara`.
- `PadThai` contains the `#noodles` tag.
- `Tiramisu` links to `Carbonara`.
- `Index.md` links to the expected recipes and contains the `#index` tag.
- The vault contains exactly five notes.

Blender: Parent-Child Rig Hierarchy

Application. `blender`

Task. Open the Blender file `/home/user/Documents/rig_hierarchy.blend`, which starts empty. Add an Empty and rename it to `Rig`. Add a cube named `Torso`, a UV sphere named `Head`, and a cylinder named `Arm`. Parent `Torso` and `Arm` to `Rig`, and parent `Head` to `Torso`, making `Head` a grandchild of `Rig`. Save the file.

Initial files. `rig_hierarchy.blend` at `/home/user/Documents/rig_hierarchy.blend`.

Representative verification checks.

- The saved Blender file exists at the expected path.
- The object `Rig` exists and has type `EMPTY`.
- The objects `Torso`, `Head`, and `Arm` exist and have mesh object types.
- `Torso` is parented to `Rig`.
- `Arm` is parented to `Rig`.
- `Head` is parented to `Torso`.