

UniClawBench: A Universal Benchmark for Proactive Agents on Real-World Tasks

Zhekai Chen^{1*} Chengqi Duan^{1*} Kaiyue Sun^{1*} Bohao Li¹ Yuqing Wang¹
 Manyuan Zhang^{2†} Xihui Liu^{1†}
¹HKU MMLab ²Meituan

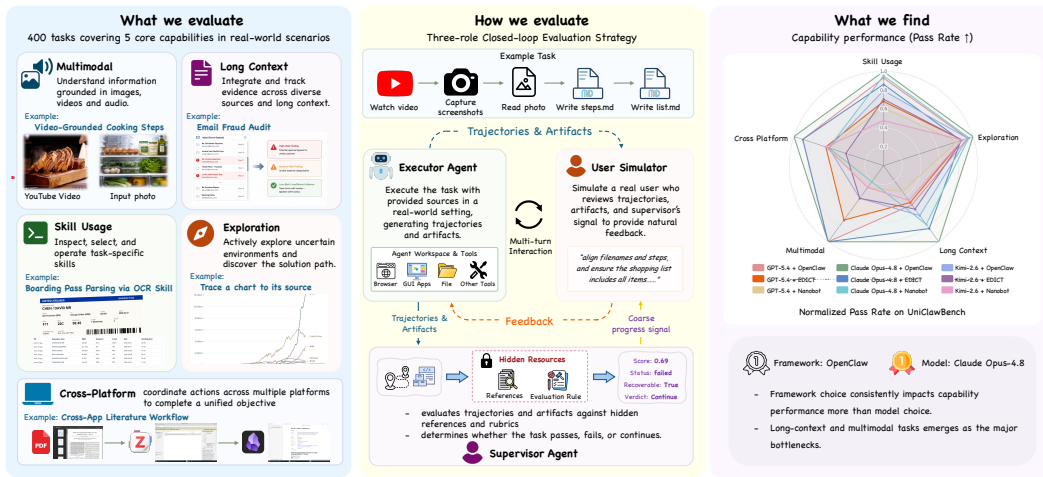


Figure 1: **Overview of UniClawBench.** UniClawBench consists of 400 bilingual real-world tasks spanning 5 core capabilities: multimodal, long-context, skill usage, exploration, and cross-platform. We propose a three-role closed-loop evaluation framework, where an executor agent performs tasks in real environments, a supervisor evaluates trajectories and artifacts using hidden rubrics, and a user simulator provides natural feedback based on executor’s performance and supervisor’s signal, enabling multi-turn interaction. We conduct two sets of experiments to evaluate cross-model and cross-framework performance. The capability-level results reveal that framework choice exerts a stronger influence than model choice.

Abstract

The rapid development of large language models and multimodal large language models has accelerated the emergence of proactive agents capable of operating everyday tools and assisting users in real-world environments. However, existing benchmarks struggle to evaluate such agents effectively, as they often rely on sandboxed environments and single-turn evaluation paradigms. Moreover, their scenario-based task taxonomies mix multiple model capabilities within the same task category, making it difficult to identify the root causes of agent failures. To address these limitations, we introduce UniClawBench, the first capability-driven benchmark designed to evaluate proactive agents in dynamic, real-world settings. UniClawBench is built around five foundational model capabilities: Skill Usage, Exploration, Long-Context Reasoning, Multimodal Understanding, and Cross-Platform Coordination. Based on these capabilities, we design 400 bilingual real-world tasks. Unlike previous benchmarks that rely on static, pre-recorded answers, our benchmark evaluates agents in live Docker containers using fine-grained, step-by-step completion checkpoints. Furthermore, we design a closed-loop evaluation

*Equal contribution, listed alphabetically

†Corresponding Author

strategy comprising an executor agent, a hidden supervisor agent, and a user agent to simulate realistic multi-turn human feedback without leaking grading criteria. To disentangle base model capabilities from framework-level design choices, we evaluate state-of-the-art models under multiple agent frameworks. Through comprehensive comparisons across both models and frameworks, we show how base model capabilities and agent framework designs jointly shape performance in real-world environments. To facilitate future research, we make our benchmark and code publicly available at <https://github.com/HKU-MMLab/UniClawBench>.

1 Introduction

Large language models and multimodal large language models have evolved from text-centric assistants into autonomous agents capable of executing complex, multi-step tasks in real-world environments [48, 36]. This evolution has given rise to proactive agents, which assist user by directly control everyday tools like browsers and terminals. Representative platforms such as OpenClaw [39] and Nanobot [35] have moved beyond isolated task execution toward 24-7 personal assistance, effectively serving as digital coworkers that handle the full spectrum of daily work. These agents are tasked with navigating complex, real-world scenarios that span multiple modalities and platforms. This rapid evolution raises a pressing question: How can we systematically measure their true capabilities in real-world settings?

Significant efforts have been devoted to building agent benchmarks [20, 26]. However, we identify three structural limitations that prevent them from adequately evaluating modern proactive agents. First, existing benchmarks struggle to capture real-world complexity. They often rely on self-hosted website mirrors (e.g., WebArena [53]) or cached pages within virtual machines (e.g., OSWorld [45]), creating a profound gap between sandboxed performance and actual real-world capabilities. Second, they mostly leverage a single-turn evaluation paradigm, completely ignoring the closed-loop nature of human-agent interaction where users frequently provide iterative feedback. Finally, and most critically, existing benchmarks organize tasks by application scenario (e.g., “office”, “research”). This approach conflates fundamentally different abilities. When a model fails an “office” task, it is unclear whether the bottleneck lies in visual perception, long-context reasoning, or tool usage. Consequently, a capability-driven benchmark is urgently needed to pinpoint the precise source of failure.

However, building a benchmark that operates in real-world environments and incorporates closed-loop user interaction introduces two fundamental challenges. The first is the lack of a stable ground truth. The expected outputs in sandboxed benchmarks are fixed when the benchmark is created, but this assumption becomes fragile in unpredictable real-world scenarios. For example, an Amazon product page may list the same product at \$199 today and \$179 tomorrow, making a pre-recorded answer quickly outdated. The second challenge lies in faithfully simulating user interaction without compromising evaluation reliability. To create a faithful closed-loop environment, an automated user simulator must dynamically provide natural corrections based on the agent’s intermediate outputs. However, if this simulator is granted access to the underlying ground truth or grading criteria to guide the agent, it risks inadvertently leaking the solution or evaluation rules to the agent under test. Balancing natural, multi-turn human feedback with strict information isolation to prevent data contamination remains a significant technical challenge.

In this paper, we introduce UniClawBench, the first capability-driven benchmark designed to evaluate proactive agents in dynamic real-world environments. UniClawBench systematically decomposes tasks along five foundational capability dimensions: Skill Usage, Exploration, Long Context, Multimodal, and Cross-Platform. To ensure comprehensive coverage, we manually design 400 bilingual tasks (English and Chinese) that encompass a wide range of authentic daily usage scenarios. All tasks run inside Docker containers equipped with real software, live browsers and local file systems. UniClawBench introduces two novel assessment mechanisms. First, to address the dynamic nature of live environments where stable ground truth is absent, we abandon pre-recorded answers. Instead, we design fine-grained, step-by-step completion checkpoints for each task. This empowers an automated supervisor with hidden evaluation criteria, ensuring it knows precisely which intermediate reasoning and execution evidence to assess at each step. Second, to solve the information-isolation paradox, UniClawBench implements a strict three-role closed-loop evaluation that acts as an information firewall. A hidden supervisor privately checks the agent’s progress, while a user simulator responds only to the visible trajectory and a coarse progress signal. This setup lets UniClawBench capture realistic multi-turn interactions without revealing the underlying answers or grading criteria.

We conduct two groups of experiments to benchmark cross-model and cross-framework performance. First, we evaluate a wide range of state-of-the-art models under OpenClaw framework, which allows us to isolate differences in base model capabilities. Second, we evaluate representative models across three agent frameworks: OpenClaw [39], EDICT [6] and Nanobot [35]. This allows us to study how framework design affect performance on different tasks. Together, these comparisons reveal not only which models are more capable, but also how different agent architectures amplify or limit these capabilities in complex real-world tasks.

In summary, our main contributions are as follows:

- We introduce **UniClawBench**, the first capability-driven benchmark that evaluates proactive agents in dynamic real-world environments. UniClawBench is built around five foundational capabilities, based on which we manually design 400 bilingual tasks to diagnose the root causes of agent failures.
- We propose a closed-loop evaluation strategy that better reflects real-world user-agent interaction. It introduces a three-role design with an executor agent, a hidden supervisor agent, and a user simulator agent, enabling multi-turn feedback while preventing answer or grading-criteria leakage.
- We evaluate state-of-the-art models across multiple agent frameworks. This joint analysis reveals how base model capabilities and framework designs each contribute to agent performance across different capability dimensions.

2 Related Work

Large language models as autonomous agents. Recent progress in large language models has enabled autonomous agents that interleave reasoning, action, observation, and tool use [7, 11, 31, 32, 42, 47]. ReAct [48] established a representative reasoning–acting loop, while Toolformer [36] showed that language models can learn to invoke external tools. Subsequent systems extended this paradigm toward richer orchestration and longer-horizon problem solving: HuggingGPT [37] uses an LLM as a controller over specialized models, ToolLLM [33] studies large-scale API selection and invocation, and Reflexion [38] improves agents through verbal feedback and memory. Beyond text-only interaction, GUI and computer-use agents such as CogAgent [15] and AppAgent [51] extend the paradigm to screens and mobile interfaces. These works establish the basic agent loop and tool-use patterns, but they mainly target generic autonomy rather than persistent personal assistance.

Proactive agent systems. Proactive agents, distinct from reactive agents, initiate actions and pursue goals autonomously, a concept formalized by [43]. This foundational work underpins modern long-running assistants that leverage persistent memory, reusable skills, and local tool access. OpenClaw [39] offers a robust hub-and-spoke architecture for multi-platform integration. Nanobot [35] emphasizes an ultra-lightweight, extensible core for research and easy modification. Hermes Agent [27] features a built-in learning loop for autonomous skill creation and continuous self-improvement, building a deeper user model across sessions. These systems collectively highlight a shared design space for individualized assistance through advanced memory, skills, and tool management.

Benchmark for AI Agents. Earlier benchmarks evaluate LLM agents across web, OS, mobile, and enterprise environments [8, 10, 17, 20, 22, 26, 34, 45, 46, 49, 53], but they target isolated task execution rather than proactive personal assistance. More recent benchmarks instead targets proactive agent systems and falls into three groups. The first group measures end-to-end task completion in real environments. For instance, ClawBench [52] evaluates write-heavy actions on live production websites to expose the gap between sandbox and real-world performance, while related efforts tackle harder OpenClaw tasks and desktop applications [21, 1, 16]. The second group studies skill use and adaptation. This is exemplified by SkillsBench [19], which contrasts curated and self-generated skills to quantify their marginal effects, as well as subsequent research on continual skill learning [44, 23]. The third group emphasizes trustworthiness and dynamic environments: ClawMark [25] models multi-day coworker scenarios with evolving backends, while other works formalize adversarial attacks and trajectory-aware evaluation [41, 50]. Despite this progress toward live execution, evaluating agents in real-world environments introduces critical instability, yet these recent benchmarks still rely on static ground-truth answers or single-turn evaluation. Furthermore, they organize tasks by scenario and lack analysis on agent frameworks. In contrast, UniClawBench tackles dynamic environments via hidden checkpoint rubrics and a closed-loop multi-turn evaluation strategy. Moreover, its capability-oriented taxonomy and cross-framework evaluation (OpenClaw, Nanobot, EDICT) explicitly disentangle intrinsic model competence from framework-level design choices.

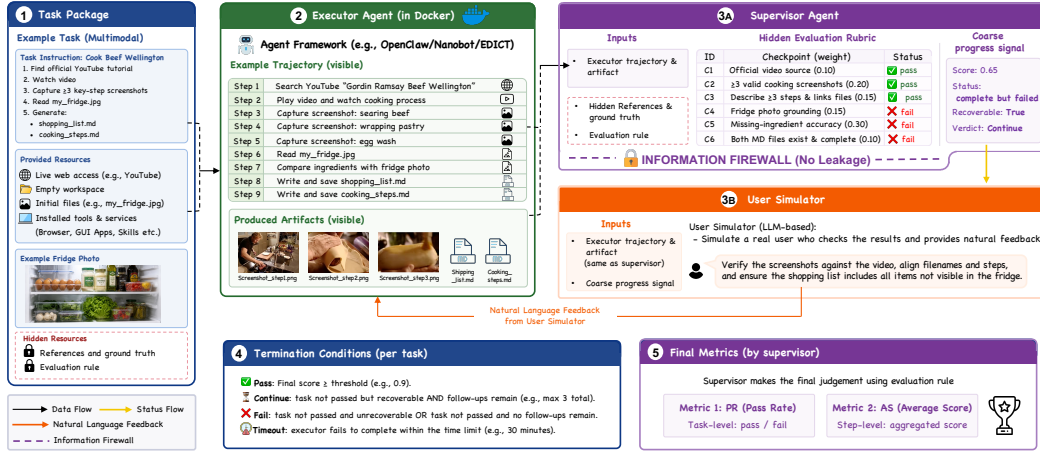


Figure 2: **Three-role Closed-loop Evaluation Strategy.** A task is executed by an agent (Executor) within a real-world environment using only public inputs (e.g., tools, files, and web access). The resulting observable trajectory and produced artifacts are assessed by a hidden Supervisor, which applies fine-grained checkpoint-based rubrics and private references to compute a structured evaluation state (e.g., pass / fail / continue) and score. To prevent leakage of evaluation criteria, only a coarse progress signal is exposed through an information firewall to a User Simulator, which generates natural-language feedback based solely on visible evidence. The sanitized feedback is returned to the executor for iterative refinement, forming a multi-turn closed loop. The example illustrates a multimodal task (Beef Wellington cooking with fridge grounding), where the agent must produce screenshots and grounded outputs, emphasizing strict evidence-based evaluation.

3 UniClawbench

This section presents the architectural framework of UniClawBench. We first establish a capability-oriented task taxonomy and detail the benchmark composition, including task categorization, the hierarchical task-package structure, and comprehensive data statistics (§3.1). Subsequently, we articulate the three-role closed-loop evaluation strategy, which decouples the evaluated executor, hidden supervisor agent, and public user simulator agent to support multi-turn feedback without leaking private grading information (§3.2). Finally, we elaborate on the execution and evaluation system, including the docker-based runtime, task-resource injection, artifact collection, and supervisor-driven scoring pipeline (§3.3).

3.1 Capability-Oriented Task Taxonomy and Benchmark Composition

Existing benchmarks typically organize tasks by application domain (e.g., shopping or travel) or by interaction surface (e.g., web and desktop environments). These taxonomies are useful for comparing performance across settings, but they provide limited diagnostic evidence about which capability causes an agent to fail. A failure in a domain-level task may reflect many different bottlenecks, such as incorrect tool use, insufficient exploration, weak visual grounding, poor long-context reasoning, or failure to coordinate across applications. In realistic personal workflows, a capable proactive agent must invoke specialized tools, investigate unfamiliar or underspecified environments, maintain consistency over long evidence chains, ground its actions in non-textual signals, and coordinate state across multiple applications. These requirements motivate UniClawBench’s capability-oriented taxonomy, which categorizes tasks into five core dimensions: Skill Usage, Exploration, Long-Context Reasoning, Multimodal Understanding, and Cross-Platform Coordination. Each task is assigned to the category that best captures its primary capability bottleneck, i.e., the capability whose absence would prevent successful completion.

Skill-usage tasks. Skill-usage tasks evaluate an agent’s proficiency in selecting, inspecting, and operating task-specific tools or APIs. The task package incorporates explicit skills, local services, fixture data, or required command-line workflows. The key bottleneck is therefore not discovering where information may exist, but correctly choosing, understanding, and executing the provided tool interface to produce a verifiable artifact. Representative tasks include OCR of boarding-pass scans,

spreadsheet and CSV reconciliation, access-log analysis, audio transcription, Mermaid or diagram generation, SQLite querying, Git or Docker audits, document conversion, and API-backed workflows. These tasks are designed so that prior knowledge or a plausible final answer is insufficient: the agent must actually consult the declared skill or tool interface, apply it to the provided data, and save structured outputs such as JSON, CSV, reports, logs, or converted files.

Exploration tasks. Exploration tasks characterize an agent’s capacity for open-ended investigation in the presence of incomplete, noisy, or potentially misleading information. Agents are required to reverse-engineer undocumented web APIs and their real fields to pull data instead of scraping HTML, audit shell or infrastructure configurations and reconcile silent conflicts, select a single best option from a fixed candidate set under quantitative constraints, trace images or data points back to their authoritative source, validate tool behavior inside local fixtures, or solve single-position puzzles and board games. Unlike skill-usage tasks, where the relevant tool interface is usually explicit, exploration tasks require the agent to determine which sources, files, services, logs, or hypotheses are relevant before it can complete the task. These tasks emphasize the process of exploration itself: agents must examine candidate sources, reject incorrect alternatives, document negative evidence, and leave a trace that distinguishes real investigation from a guessed conclusion. The expected outputs usually include both a final artifact and supporting audit records, such as logs, JSON notes, command transcripts, screenshots, hashes, or source comparisons.

Long-context Reasoning tasks. Long-context tasks evaluate whether agents can aggregate and maintain evidence across heterogeneous sources, long documents, or extended interaction traces. The defining difficulty is not the length of any single document, but the need to preserve constraints, reconcile conflicting evidence, and produce a globally consistent synthesis across many pieces of context. These tasks necessitate synthesizing information from multiple web pages, videos, PDFs, reviews, emails, logs, or research materials before producing a coherent final answer. Examples include comparing headphone recommendations across Bilibili and YouTube and then checking product prices and specifications, planning subscriptions or travel under many constraints, synthesizing reading packs into reports or slides, monitoring event streams, auditing long email or fraud records, and solving tasks that require sustained state tracking such as games or multi-step investigations. The main difficulty is not retrieving isolated facts but maintaining global consistency across scattered evidence, resolving conflicts, and producing an auditable synthesis.

Multimodal Understanding tasks. Multimodal tasks require agents to extract, interpret, and generate information grounded in images, videos, and audio. A task is assigned to this category only when the correctness of the final output depends on evidence that must be obtained from non-textual content, rather than from metadata, filenames, or plausible textual inference alone. The tasks are constructed around realistic content rather than synthetic visual questions. For example, agents may need to locate a specific figure in an academic paper and recreate it, reconstruct visual diagrams or SVG-like objects, inspect food or product photos, extract evidence from YouTube videos, generate subtitles or lyrics, or organize local image collections. These tasks typically require agents to combine visual perception with tool use and content generation: a successful run must not only identify the right visual content, but also save verifiable outputs such as reproduced plots, screenshots, scripts, captions, or structured summaries.

Cross-platform Coordination tasks. Cross-platform tasks require agents to synchronize and orchestrate information across heterogeneous applications and interfaces. We treat cross-platform coordination as a capability rather than an environment label: the agent must preserve state, transfer information, and verify side effects across application boundaries. Unlike tasks that can be completed entirely in a browser or terminal, these tasks may involve moving information among web pages, desktop GUI applications, local files, calendars, spreadsheets, note-taking tools, citation managers, PDF readers, and communication platforms. For example, an agent may need to read a PDF, create a Zotero record, export a BibTeX citation, write an Obsidian literature note, and save screenshots proving that the real desktop applications were used. Other tasks involve travel planning across official websites and map services, receipt reconciliation through GUI tools, calendar/event creation, or multi-application research workflows. Success depends on preserving state and evidence across platforms, not merely producing a textual answer.

Overall, UniClawBench contains 400 bilingual real world tasks, with 40 English and 40 Chinese tasks for each capability. All tasks are meticulously manually constructed based on genuine, day-to-day scenarios encountered by everyday users. We intentionally constrain and design each scenario so that its completion primarily bottlenecks on a single model capability, while allowing auxiliary operations to appear naturally when needed. Each task is designed as a complete task package, rather than a standalone prompt-answer pair. A task package includes the user instruction, task-specific context,

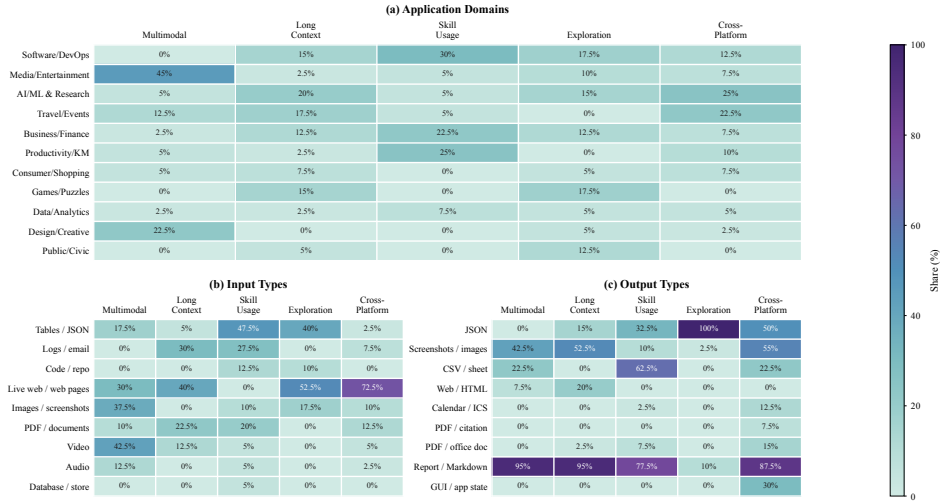


Figure 3: Diversity and composition statistics of UniClawBench. The heatmaps detail (a) the broad application-domain coverage designed to prevent scenario bias, as well as the diverse (b) input and (c) output formats. These distributions reflect the multimodal and heterogeneous nature of real-world agent tasks. Each value denotes the within-category percentage.

input files or web resources, tools and services, skills, expected outputs, and hidden evaluation references. A representative task example is illustrated in Figure 2.

To ensure a comprehensive and realistic evaluation, UniClawBench encompasses a highly diverse set of task formats and scenarios. As detailed in the Figure 3, our benchmark spans a broad spectrum of application domains, ensuring that the evaluation of each foundational capability is not biased toward any specific domain. Furthermore, to faithfully reflect the heterogeneous nature of real-world tasks, the tasks incorporate a rich variety of input and output modalities. Rather than relying exclusively on standard text or JSON, agents are required to process complex inputs, such as live web pages, code repositories, videos, and databases, and successfully produce diverse, actionable outputs, including calendar files, office documents, diagrams, and GUI application states.

3.2 Three-role Closed-loop Evaluation Strategy

Unlike single-turn assistants, proactive agents are typically used through multi-turn user-agent interaction. A user may inspect an intermediate result, point out missing requirements, ask for refinement, or provide additional constraints. Therefore, evaluating only the first response does not fully capture whether an agent can recover from partial progress and continue a task under user feedback. UniClawBench introduces a closed-loop evaluation design that models this interaction while preserving controlled and reproducible grading. The central challenge is to provide useful follow-up feedback without exposing the hidden evaluation criteria.

Our design involves three roles. The **executor** is the agent being evaluated. It receives only the public task instruction and the permitted task package, including available files, tools, services, and runtime resources. The **supervisor** is a hidden evaluator agent. It observes the full execution trajectory and the produced artifacts, autonomously analyzes them against task-specific hidden references and evaluation rubrics, and gives out judgments. The **user simulator** is also an agent. It analyzes the visible execution trajectory and outputs to generate natural feedback that the executor can act on in each round. Unlike the supervisor, the user simulator is designed to emulate the behavior of a typical end user: it focuses on visible progress, result quality, and apparent issues, rather than relying on predefined references or detailed technical inspection.

This design introduces a risk of information leakage: if the user simulator observes hidden rubrics or ground-truth answers, its feedback may reveal privileged hints to the executor. UniClawBench therefore isolates the supervisor from the user simulator. Although the supervisor has access to the exact grading criteria, it only forwards a limited high-level status signal, such as whether the attempt has passed, failed, or remains recoverable. Relying only on this status signal and the agent’s visible trajectory, the user simulator generates natural feedback without access to hidden references or supervisor rationales. A feedback rewriter further sanitizes this feedback before it is delivered

to the executor. This workflow mimics realistic human-agent interaction while keeping the hidden evaluation criteria separated from the agent-facing dialogue. An illustration is shown in Figure 2.

3.3 Evaluation System

UniClawBench implements the closed-loop evaluation through a host-supervised, container-executed runtime. For each task, the executor is launched in a fresh Docker environment containing the public instruction, input sources, tools, skills and relevant services. Hidden references and grading rubrics are never exposed to the executor. The executor runs inside the container using one of the supported agent systems, such as OpenClaw, Nanobot, or EDICT. It can interact with the browser, command line, GUI applications, local files, and injected services, and is instructed to save useful evidence and final outputs to a designated result directory. After each executor turn, the runner collects the observable trajectory, including transcripts, tool-use records, runtime states, and saved artifacts. These materials are then provided to the hidden supervisor together with the task-specific evaluation references. The supervisor works in a separate workspace and it scores the executor’s performance using checkpoint-based rubrics and returns a structured evaluation state: pass, fail, or continue. A pass indicates that the task has been successfully completed, while a fail denotes a fundamental, irrecoverable error where the objective can no longer be achieved. A continue signifies that the agent’s current progress is flawed or incomplete, but the task remains recoverable in subsequent turns. Alongside this status, the supervisor outputs a detailed justification, mapping the collected evidence to the rubrics to substantiate its decision. If the attempt is incomplete but recoverable, the public user simulator receives only the visible trajectory and the high-level status signal. It generates natural follow-up feedback, which is sanitized before being sent back to the executor as the next user message. The loop continues until the task passes, fails irrecoverably, reaches the follow-up budget, or encounters a timeout limit. This system preserves realistic multi-turn interaction while keeping grading information isolated from the evaluated agent.

4 Experiments

4.1 Environmental Setup

We conduct two groups of experiments. First, we compare ten executor models under the same OpenClaw framework (v2026.3.11), which isolates model-level capability differences. Second, we select three representative models and evaluate them under OpenClaw (v2026.3.11), Nanobot (v0.1.5.post3), and EDICT, allowing us to study how agent-framework design affects certain performance. All compared runs use the same tasks, hidden evaluation rubrics and scoring pipeline. For both supervision and user simulation, we use separate Codex agents based on GPT-5.4 with high reasoning effort. To reduce context overhead, screenshots from the executor trajectory are not directly inserted into the initial context of the supervisor or user simulator. Instead, they are stored as observable files and can be inspected on demand when visual evidence is needed. To ensure a fair comparison and enable the system image to handle browser and GUI tasks, we abandon the OpenClaw browser extension, installing and preparing each frameworks with the same basic skills, including self-written Advanced Packaging Tool (apt) skill, DuckDuckGo Search skill [9], Web Search skill [5], Agent-Browser skill [18], and Desktop Control [24] skill. All experiments are executed on Intel Core i7-13700 host machines with 16GB of RAM, with every isolated Docker containers allocated 2GB of memory. On average, each task run takes approximately 17.4 minutes.

In all experiments, each task allows up to two follow-up user interactions after the initial instruction. To support the agentic and interactive nature of our tasks while avoiding unbounded execution, we impose two time limits. The first is a global timeout, which bounds the total wall-clock time across all turns of a task. The second is a per-turn timeout, which limits the duration of any single executor turn. For standard tasks, we set the global timeout to 30 minutes and the per-turn timeout to 20 minutes. For long context tasks, which require processing substantially larger inputs and performing more extensive reasoning, we increase these limits to 45 minutes and 30 minutes, respectively. We report two metrics: Pass Rate (PR), the percentage of tasks that reach a final pass state, and Average Score (AS), the mean checkpoint-based score assigned by the supervisor. When a run exceeds either the global or per-turn timeout, we credit the task with the highest checkpoint-based score achieved among its completed turns, rather than discarding the partial progress made up to that point.

4.2 Benchmark Evaluation Reliability Study

To validate the reliability of our automatic evaluation pipeline, we randomly sample 50 completed trajectories and ask three human experts to independently evaluate each of them. Each expert provides both a binary pass/fail judgment and a continuous completion score corresponding to PR and AS, respectively. For human PR, we take the majority vote among the three experts and for human AS, we average their completion scores. We then compare these aggregated human judgments with the supervisor-produced PR and AS. The automatic pass/fail decision achieves 92.0% agreement with the human majority vote, while the checkpoint-based AS obtains a strong correlation with averaged human completion scores, with Pearson $r = 0.71$ and Spearman $\rho = 0.68$.

Table 1: Average quantitative evaluation across English and Chinese subsets of UniClawBench. To isolate inherent model capabilities from framework-level design choices, all 10 models are evaluated as the executor under the same OpenClaw framework. Performance is measured by Pass Rate (PR) and checkpoint-based Average Score (AS) across the five foundational capability categories.

Model	Skill Usage		Exploration		Long Context		Multimodal		Cross Platform		Overall	
	PR	AS	PR	AS	PR	AS	PR	AS	PR	AS	PR	AS
Kimi-2.6 [2]	0.438	0.834	0.775	0.884	0.287	0.655	0.075	0.494	0.237	0.679	0.362	0.709
Qwen-3.5-Plus [40]	0.300	0.771	0.787	0.915	0.200	0.648	0.075	0.583	0.225	0.737	0.318	0.731
GPT-4.1 [28]	0.075	0.492	0.512	0.769	0.075	0.338	0.075	0.418	0.025	0.425	0.152	0.488
GPT-5.4-Mini [30]	0.188	0.681	0.688	0.834	0.100	0.498	0.062	0.454	0.037	0.615	0.215	0.616
Gemini-3.0-Flash [12]	0.287	0.750	0.800	0.899	0.163	0.621	0.087	0.544	0.075	0.678	0.282	0.698
Gemini-3.1-Flash-Lite [13]	0.050	0.569	0.600	0.835	0.062	0.374	0.037	0.409	0.025	0.530	0.155	0.543
Claude Sonnet-4.6 [4]	0.512	0.873	0.812	0.913	0.400	0.649	0.212	0.626	0.338	0.753	0.455	0.763
Gemini-3.1-Pro [14]	0.300	0.769	0.850	0.947	0.188	0.646	0.163	0.583	0.125	0.690	0.325	0.727
GPT-5.4 [29]	0.512	0.845	0.775	0.897	0.225	0.705	0.175	0.617	0.350	0.804	0.407	0.774
Claude Opus-4.8 [3]	0.550	0.829	0.825	0.928	0.438	0.576	0.175	0.493	0.388	0.686	0.475	0.702

4.3 Cross-Model Benchmark Results

We benchmark 10 SOTA models with Openclaw framework on the English and Chinese subset of UniClawbench. The results are presented in Table 1. The results show that while leading closed-source models like Claude Opus-4.8 and GPT-5.4 achieve the highest overall pass rates, the absolute success rates remain strictly below 50%. This highlights the extreme difficulty of the benchmark, revealing a profound gap between sandboxed capabilities and real-world task execution. Notably, open-source models are rapidly closing this performance gap. Models like Qwen-3.5-Plus and Kimi-2.6 show highly competitive results that even surpasses closed-source models like Gemini-3.1-pro. A widespread execution issue across all evaluated models is the severe “halfway failure” phenomenon, evidenced by the significant gap between their intermediate average scores and final pass rates. Most models consistently secure high checkpoint-based scores but ultimately fail to complete the entire task. This indicates that while current agents can successfully make partial progress, they frequently make irrecoverable errors during extended execution chains, lacking the reliability required for complex real-world tasks. From the capability perspective, models perform relatively better on Skill Usage and Exploration tasks, where the main challenge is to identify the right tool or evidence source and apply it correctly. In contrast, Long Context, Multimodal, and Cross Platform tasks remain much harder, because they require agents to keep memories through long trajectories, ground decisions in non-textual evidence, and coordinate actions across multiple applications. This suggests that current agents are already reasonably capable of local tool operation and information seeking, but still struggle with long-horizon memory, multimodal grounding, and cross-application coordination.

4.4 Cross-Framework Benchmark Results

Table 2 details the performance of representative models across the OpenClaw, Nanobot, and EDICT frameworks. The results reveal that framework architecture which determines how execution trajectories are organized, profoundly influences task success. The performance gap between frameworks widens as base model capabilities increase, highlighting how structural design either amplifies or bottlenecks intrinsic model reasoning.

OpenClaw consistently achieves the highest pass rate across all tested models. As a centralized, single-agent framework, its primary advantage lies in minimal information loss. The original task constraints, intermediate tool execution evidence, and closed-loop user feedback are seamlessly preserved within a unified trajectory. This cohesive context management allows strong models to reliably translate partial progress into complete, strictly verifiable task successes.

EDICT exhibits a distinct pattern: relatively high average scores coupled with notably lower pass rates. This discrepancy exposes the “coordination friction” inherent in its multi-agent orchestration.

Table 2: Average cross-framework performance across the English and Chinese subsets of UniClawBench. We evaluate three representative models across OpenClaw, EDICT, and Nanobot frameworks. Results are presented as Pass Rate (PR), Average Score (AS), and average executor-side input/output token (Token_I & Token_O) usage per task. Input tokens are reported in millions and output tokens in thousands.

Model	Framework	Skill Usage		Exploration		Long Context		Multimodal		Cross Platform		Overall			
		PR	AS	PR	AS	PR	AS	PR	AS	PR	AS	Token _I	Token _O		
GPT-5.4 [29]	OpenClaw [39]	0.512	0.845	0.775	0.897	0.225	0.705	0.175	0.617	0.350	0.804	0.407	0.774	1.15	11.0
	EDICT [6]	0.375	0.791	0.725	0.883	0.212	0.693	0.125	0.559	0.250	0.792	0.338	0.744	1.68	18.3
	Nanobot [35]	0.338	0.684	0.675	0.819	0.150	0.569	0.050	0.499	0.237	0.630	0.290	0.640	0.57	9.5
Claude Opus-4.8 [3]	OpenClaw [39]	0.550	0.829	0.825	0.928	0.438	0.576	0.175	0.493	0.388	0.686	0.475	0.702	0.78	16.4
	EDICT [6]	0.475	0.751	0.713	0.838	0.362	0.583	0.175	0.576	0.350	0.687	0.415	0.687	2.15	42.7
	Nanobot [35]	0.525	0.766	0.787	0.867	0.338	0.548	0.037	0.264	0.237	0.488	0.385	0.587	0.49	16.2
Kimi-2.6 [2]	OpenClaw [39]	0.438	0.834	0.775	0.884	0.287	0.655	0.075	0.494	0.237	0.679	0.362	0.709	1.09	22.9
	EDICT [6]	0.388	0.813	0.725	0.868	0.250	0.601	0.075	0.506	0.163	0.686	0.320	0.695	2.53	51.5
	Nanobot [35]	0.263	0.684	0.725	0.847	0.200	0.501	0.050	0.317	0.150	0.519	0.278	0.573	0.86	21.3

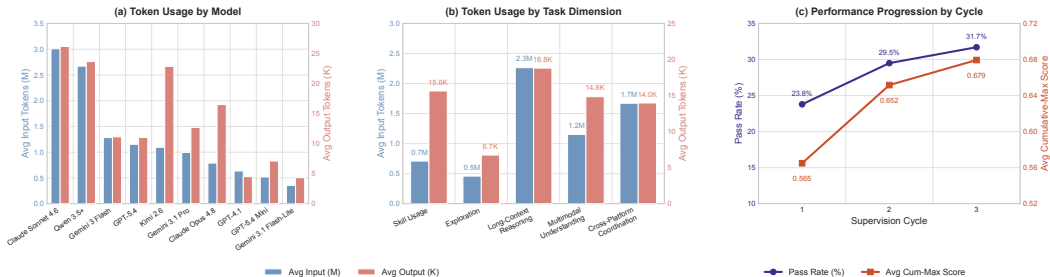


Figure 4: **Token Usage and Performance Progression by Cycle.** (a) and (b) display the token usage on the OpenClaw system, categorised by model and task dimension, (c) demonstrates the performance progression by cycle motivated by the user follow-ups.

The central orchestrator dispatches tasks based on discrete state polling (e.g., Kanban boards) but lacks real-time supervision over downstream sub-agents. If a sub-agent fails to transmit precise state updates or drops context during handoffs, the execution pipeline stalls, yielding partial progress that fails. Furthermore, during long-context tasks, identity and role constraints are easily forgotten. As a result, the orchestrator does the work itself instead of assigning it. The system pays the high token cost of multiple agents but gets none of the collaborative benefits.

Designed with an ultra-lightweight structure, Nanobot demonstrates a highly optimized token usage in practice, utilizing significantly fewer tokens than OpenClaw (e.g., 0.57M vs. 1.15M average input tokens for GPT-5.4). However, this extreme efficiency introduces an inherent trade-off with performance. In rigorous, real-world tasks requiring extended evidence chains and complete trajectories, this simplified context management often struggles to produce fine-grained reasoning steps and long textual evidence. Consequently, while the framework effectively achieve partial progress and secures high intermediate checkpoint scores, it frequently fails to generate the complete results demanded by strict evaluations, resulting in a notably lower overall pass rates.

4.5 Token Usage and Performance Progression

As illustrated in Figure 4 (a) (b), under the OpenClaw framework, token usage varies significantly across different models and inherently increases for capability dimensions where current models exhibit relative weaknesses, like long-context and multimodal understanding. Table 2 also provides statistics on token usage across different frameworks, where EDICT consume a huge amount of token in total and Nanobot keeps its extreme token efficiency. Furthermore, Figure 4 (c) demonstrates that agent performance consistently improves across interaction cycles, validating that multi-turn user feedback is essential for dynamic error recovery. This also mimics user-agent interaction in real life.

5 Conclusion and Limitations

In this paper, we introduce UniClawBench, a capability-driven benchmark designed to evaluate proactive agents in dynamic, real-world environments. Moving beyond static sandboxes, we decompose agent evaluation into five foundational capability categories across 400 bilingual real-world tasks. To

faithfully capture multi-turn human-agent collaboration without leaking grading criteria, we propose a novel three-role closed-loop framework comprising an executor agent, a hidden supervisor agent, and a user simulator agent. Our evaluations reveal that current agents struggle with long-horizon memory and cross-platform coordination, and that framework architectures often bottleneck intrinsic model capabilities. While UniClawBench introduces a rigorous evaluation framework, it is limited by the relatively small set of 400 manually curated tasks, possible instability from live-environment execution, and the potential bias that may be introduced by LLM-based evaluation. Despite these constraints, UniClawBench successfully pinpoints the root causes of agent failures, paving the way for more reliable proactive assistants.

References

- [1] Kilo AI. Pinchbench: Real-world benchmarks for ai coding agents. <https://github.com/pinchbench/skill>, 2026.
- [2] Kimi AI. Kimi k2.6: From code to creation, from one to many. <https://www.kimi.com/ai-models/kimi-k2-6>, 2026.
- [3] Anthropic. Introducing claude opus 4.6. <https://www.anthropic.com/news/claude-opus-4-6>, 2026.
- [4] Anthropic. Introducing claude sonnet 4.6. <https://www.anthropic.com/news/claude-sonnet-4-6>, 2026.
- [5] Billyutw. Openclaw web-search skill. <https://clawhub.ai/billyutw/web-search>, 2026.
- [6] cft0808. Openclaw edict. <https://github.com/cft0808/edict>, 2026.
- [7] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.
- [8] De Chezelles, Thibault Le Sellier, Sahar Omid Shayegan, Lawrence Keunho Jang, Xing Han Lù, Ori Yoran, Dehan Kong, Frank F Xu, Siva Reddy, Quentin Cappart, et al. The browsergym ecosystem for web agent research. *arXiv preprint arXiv:2412.05467*, 2024.
- [9] Cscsxx606. Duckduckgo search openclaw. <https://github.com/cscsxx606/duckduckgo-search-openclaw>, 2026.
- [10] 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.
- [11] Chengqi Duan, Kaiyue Sun, Rongyao Fang, Manyuan Zhang, Yan Feng, Ying Luo, Yufang Liu, Ke Wang, Peng Pei, Xunliang Cai, et al. Codeplot-cot: Mathematical visual reasoning by thinking with code-driven images. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9586–9596, 2026.
- [12] Google. Gemini 3 flash. <https://deepmind.google/models/gemini/flash/>, 2025.
- [13] Google. Gemini 3.1 flash lite. <https://deepmind.google/models/gemini/flash-lite/>, 2026.
- [14] Google. Gemini 3.1 pro. <https://deepmind.google/models/gemini/pro/>, 2026.
- [15] Wenyi Hong, Weihang Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, et al. Cogagent: A visual language model for gui agents. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 14281–14290, 2024.
- [16] JetAstra. Macagentbench: Benchmark agents where they actually work — on macos. <https://github.com/JetAstra/MacAgentBench>, 2026.

- [17] 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)*, pages 881–905, 2024.
- [18] Vercel Labs. Agent browser openclaw. <https://github.com/vercel-labs/agent-browser>, 2026.
- [19] Xiangyi Li, Wenbo Chen, Yimin Liu, Shenghan Zheng, Xiaokun Chen, Yifeng He, Yubo Li, Bingran You, Haotian Shen, Jiankai Sun, et al. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026.
- [20] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.
- [21] Xiang Long, Li Du, Yilong Xu, Fangcheng Liu, Haoqing Wang, Ning Ding, Ziheng Li, Jianyuan Guo, and Yehui Tang. Liveclawbench: Benchmarking llm agents on complex, real-world assistant tasks. *arXiv preprint arXiv:2604.13072*, 2026.
- [22] Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. Agentboard: An analytical evaluation board of multi-turn llm agents. *Advances in neural information processing systems*, 37:74325–74362, 2024.
- [23] Ziyu Ma, Shidong Yang, Yuxiang Ji, Xucong Wang, Yong Wang, Yiming Hu, Tongwen Huang, and Xiangxiang Chu. Skillclaw: Let skills evolve collectively with agentic evolver. *arXiv preprint arXiv:2604.08377*, 2026.
- [24] Matagul. Desktop control openclaw. <https://clawhub.ai/matagul/desktop-control>, 2026.
- [25] Fanqing Meng, Lingxiao Du, Zijian Wu, Guanzheng Chen, Xiangyan Liu, Jiaqi Liao, Chonghe Jiang, Zhenglin Wan, Jiawei Gu, Pengfei Zhou, et al. Clawmark: A living-world benchmark for multi-turn, multi-day, multimodal coworker agents. *arXiv preprint arXiv:2604.23781*, 2026.
- [26] Grégoire Mialon, Clémentine Fourier, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*, 2023.
- [27] NousResearch. Hermes agent: An agent that grows with you. <https://github.com/nousresearch/hermes-agent>, 2026.
- [28] OpenAI. Introducing gpt-4.1 in the api. <https://openai.com/index/gpt-4-1/>, 2025.
- [29] OpenAI. Introducing gpt-5.4. <https://openai.com/index/introducing-gpt-5-4/>, 2026.
- [30] OpenAI. Introducing gpt-5.4 mini and nano. <https://openai.com/zh-Hans-CN/index/introducing-gpt-5-4-mini-and-nano/>, 2026.
- [31] Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*, 2023.
- [32] Aaron Parisi, Yao Zhao, and Noah Fiedel. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255*, 2022.
- [33] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*, 2023.
- [34] Christopher Rawles, Sarah Clinckemahillie, 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.

- [35] Xubin Ren et al. Nanobot: The ultra-lightweight personal ai agent. <https://github.com/HKUDS/nanobot>, 2026.
- [36] 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. *Advances in neural information processing systems*, 36: 68539–68551, 2023.
- [37] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36:38154–38180, 2023.
- [38] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.
- [39] Peter Steinberger. Openclaw. <https://github.com/openclaw/openclaw>, 2025.
- [40] Qwen Team. Qwen3. 5-omni technical report. *arXiv preprint arXiv:2604.15804*, 2026.
- [41] Yuhang Wang, Feiming Xu, Zheng Lin, Guangyu He, Yuzhe Huang, Haichang Gao, Zhenxing Niu, Shiguo Lian, and Zhaoxiang Liu. From assistant to double agent: Formalizing and benchmarking attacks on openclaw for personalized local ai agent. *arXiv preprint arXiv:2602.08412*, 2026.
- [42] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.
- [43] Michael Wooldridge and Nicholas R Jennings. Intelligent agents: Theory and practice. *The knowledge engineering review*, 10(2):115–152, 1995.
- [44] Peng Xia, Jianwen Chen, Xinyu Yang, Haoqin Tu, Jiaqi Liu, Kaiwen Xiong, Siwei Han, Shi Qiu, Haonian Ji, Yuyin Zhou, et al. Metaclaw: Just talk—an agent that meta-learns and evolves in the wild. *arXiv preprint arXiv:2603.17187*, 2026.
- [45] 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.
- [46] Frank F Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, et al. Theagentcompany: benchmarking llm agents on consequential real world tasks. *arXiv preprint arXiv:2412.14161*, 2024.
- [47] Zhengyuan Yang, Linjie Li, Jianfeng Wang, Kevin Lin, Ehsan Azarnasab, Faisal Ahmed, Zicheng Liu, Ce Liu, Michael Zeng, and Lijuan Wang. Mm-react: Prompting chatgpt for multimodal reasoning and action. *arXiv preprint arXiv:2303.11381*, 2023.
- [48] 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 *The eleventh international conference on learning representations*, 2022.
- [49] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. tau-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.
- [50] Bowen Ye, Rang Li, Qibin Yang, Yuanxin Liu, Linli Yao, Hanglong Lv, Zhihui Xie, Chenxin An, Lei Li, Lingpeng Kong, et al. Claw-eval: Toward trustworthy evaluation of autonomous agents. *arXiv preprint arXiv:2604.06132*, 2026.
- [51] Chi Zhang, Zhao Yang, Jiakuan Liu, Yanda Li, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. Appagent: Multimodal agents as smartphone users. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, pages 1–20, 2025.
- [52] Yuxuan Zhang, Yubo Wang, Yipeng Zhu, Penghui Du, Junwen Miao, Xuan Lu, Wendong Xu, Yunzhuo Hao, Songcheng Cai, Xiaochen Wang, et al. Clawbench: Can ai agents complete everyday online tasks? *arXiv preprint arXiv:2604.08523*, 2026.

- [53] 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.

A Task Definition Format

A UniClawBench task is generally defined by two files: a single task declaration written in YAML that fixes the public contract, and a hidden `eval_rule.md` that specifies the rubric. The executor sees only the task field rendered into a runtime prompt; the supervisor sees both files and treats the eval rule as the canonical scoring contract. Appendix A.1 gives the YAML template that fixes the public contract; Appendix A.2 walks through the semantics of each field; and Appendix A.3 specifies the nine-section schema of the hidden eval rule.

A.1 Task YAML Template

Task YAML Template

```
task_id: task_NNN_short_slug
category: <suite_name>
agent_sys: openclaw
agent_id: main
model: claude-opus-4.6

timeout_seconds: 1200
max_total_seconds: 1800
success_threshold: 0.90

task: |
  <natural-language request shown to the executor>

task_snapshot: |
  <parallel-safe variant>

references:
  - references/eval_rule.md
  - references/ground_truth.json
  - ...

sources:
  - <relative_path>

skills:
  - <skill_id>

services:
  - name: <service>
    path: <subdir>
    start: bash install.sh
    oneshot: true

pre_exec:
  - ops/populate.py
pre_exec_parallel_safe: true

codex:
  max_user_followups: 2
  user_simulator:
    policy: |
      <override of the default user-simulator policy>
  supervisor:
    instructions: |
      <task-specific addendum to the default supervisor instructions>
```

A.2 Field Semantics

Basic definition. `task_id` is unique within the suite and is also the file stem. `category` is one of the five task dimensions from the main paper. `agent_sys`, `agent_id`, and `model` record the *default*

executor binding when the task is dispatched without an override; cross-architecture and cross-model sweeps override these at runtime.

Lifecycle. `timeout_seconds` bounds a single executor cycle. `max_total_seconds` bounds the whole-attempt wall clock across all cycles, excluding supervisor and user-simulator turns. `success_threshold` is the minimum supervisor score that records the attempt as Pass. The cycle count is bounded separately by `codex.max_user_followups`.

Task prompt. `task` is the literal natural-language request the executor receives, wrapped in a small runtime preamble that surfaces workspace paths and installed skills. `task_snapshot` is an optional snapshot version that consumes snapshot data instead of a live API.

Injections. `sources` lists files placed under the executor’s workspace mount. `skills` lists declared skills; the eval rule for a skill-usage task typically requires transcript evidence that the declared skill was actually consulted. `services` entries are docker-compose-style bootstrap services started before the executor’s first turn (databases, mock APIs, GUI app installers). `pre_exec` scripts run once after services are healthy and typically populate session-specific fixture state into live API. `references` are hidden judging assets placed under the supervisor’s workspace and never copied into the executor container.

Supervisor and user simulator. The `codex` block configures the closed-loop interaction. `max_user_followups` bounds the number of follow-up cycles allowed before the runner forcibly closes the attempt. `codex.user_simulator.policy` overrides the default user-simulator behavior policy when a task needs a non-standard interaction style (e.g. a deliberately terse user, a user with a specific persona constraint). `codex.supervisor.instructions` appends a task-specific addendum to the default supervisor template, used to highlight checkpoints that are easy to miss without explicit attention or to narrow down what counts as evidence for an unusual artifact. Both blocks are optional; when absent, the defaults from Appendix C apply.

A.3 Eval Rule (`eval_rule.md`) Format

Every task ships a hidden Markdown rubric at `references/eval_rule.md`. The rubric follows a fixed nine-section schema: §1–§4 establish the contract, §5–§6 define scoring, and §7–§9 cover lifecycle policy and asset bookkeeping. The supervisor prompt instructs the supervisor to score using the lines that appear in §5 and §6, and other parts guide the understanding and flexible judgment.

Eval rule schema

1. Grading Philosophy

States what the task tests, what is rewarded, and what should not be penalized. Notes the role of declared skills.

2. Task Contract

Restates the public prompt with deliverable paths pinned, and separates hard requirements (must produce) from soft requirements (must follow these formats).

3. Source-Selection and Target-Resolution Rules

How the supervisor maps executor artifacts to ground-truth entities – alias rules, filename-stem variants, numeric tolerance bands.

4. Ground-Truth Snapshot

Pointer to ground-truth files with canonical entity counts and any must-not-appear entities for negative checkpoints.

5. Checkpoint Rubric

Numbered scoring lines whose weights sum to 1.00. Each is boolean, graded, or count-based. The supervisor scores using only these lines.

6. Scoring Policy & Score Caps

Hard ceilings applied after the §5 sum. A cap is a maximum and cannot raise a §5 sum that is already lower.

7. Continue vs Fail Guidance

Score banding for the `pass` / `continue` / `fail` verdicts. The mid-band lets the supervisor request one focused follow-up.

8. Hidden Reference Assets

Inventories of supervisor-only files; reaffirms that none are surfaced to the executor or user simulator.

9. Dynamic Content Note

Whether the task is offline (static fixture) or live (consumes an external page that may shift between captures), with reconciliation rules for live tasks.

B Task Examples

This section lists representative task examples from each suite. For every task dimension we show two prompts. Each prompt is the literal natural-language task field that the executor sees at runtime; the surrounding YAML schema follows Appendix A.

B.1 Skill Usage

Tasks here ship several declared skills and require the executor to actually consult it. The agent's job is to discover the skill, follow its SKILL.md entry-point, and apply the right tool to the provided data.

task_05_org_chart_mermaid

Our CEO asked for a governance visualization she can drop into her board deck, and I need to pull it from the proxy statement. The excerpt lives at /tmp_workspace/clawbench/sources/alphabet_def14a_excerpt.md -- use the Mermaid diagramming skill to turn it into a clean org chart. She wants to see the actual reporting hierarchy -- the solid lines from each current director to the board and from each non-CEO executive officer to the executive roster -- who sits on which committee across the audit, compensation, governance, and executive committees, and the dotted advisory or observer relationships that aren't formal seats. We also need who chairs what, and any seats that are vacant, interim, advisor-only, or recently filled.

Please render it as Mermaid using graph TD or flowchart TD in /tmp_workspace/results/org.mmd, and produce an SVG rendering at /tmp_workspace/results/org.svg. Use solid arrows for board and executive roster relationships that are explicitly supported by the excerpt, dotted arrows for committee memberships and any non-voting advisor or observer relationships, and group each committee as a subgraph. Distinguish committee chairs visually from regular members -- shape or styling is fine. Treat each person as a single node; if someone holds multiple roles, draw their relationships from that one node rather than duplicating them. Anyone listed as recently-departed should not appear as a current node.

Also save /tmp_workspace/results/governance_matrix.csv so my chief of staff can review the extraction without reading Mermaid. Use columns: person, source_role, governance_group_or_board, committee, committee_role, edge_type, source_quote.

task_19_github_activity_digest

I missed the last two weeks of activity on clawbench-activity-fixture and need to catch up before the maintainer sync. Build me a 16-day activity digest covering 2026-04-10 through 2026-04-25 inclusive -- use the GitHub skill in our workspace for the API calls. Authenticate with the \$GITHUB_TOKEN environment variable and hit the live GitHub REST API directly via the gh CLI or curl against api.github.com. Do not rely on any local cache or pre-exported snapshot file under /tmp_workspace/clawbench for this run; everything must come from a fresh API pull covering merged PRs, closed-unmerged PRs, closed issues, and commits in that window.

A few entries in this fixture repo may have been imported by an automation account. When a PR or issue body, or a commit message, includes an explicit pseudo-user or planned-date note, use that note as the contributor and date for the digest instead of the automation account's login or the rebuild timestamp.

Save `/tmp_workspace/results/digest.md` grouped by contributor. For each contributor show: PRs merged, PRs closed without merge, issues closed, and commit count. Add a Follow-up Flags section for activity that looks risky or needs a maintainer follow-up, including every closed-unmerged PR and any revert PR or revert commit. End with a Highlights section listing the 5 most recent merged PRs or closed issues in the window, newest first. When two highlight events share the same date, list the higher-numbered activity first so number 27 comes before number 13; if the numbers also tie, list the issue before the pull request.

Also save `/tmp_workspace/results/activity_index.csv` with columns `activity_type,number_or_sha,title_or_message,contributor,activity_date,status` covering every merged PR, closed-unmerged PR, closed issue, and commit in the 16-day window so I can sort and filter it later. Cover the full set of contributors and activity, not a partial sample.

B.2 Exploration

Tasks here are open-ended audits or refactors over noisy and potentially misleading inputs. The agent must investigate multiple choices, reject incorrect alternatives, and document negative evidence to distinguish real exploration from a guessed conclusion.

task_01_zshrc_prompt_refactor

Your goal is to refactor the injected legacy ``.zshrc`` into a maintainable, low-noise, verifiable zsh configuration. The input file is at `~/tmp_workspace/sources/existing_zshrc``. This is not about writing a new prompt snippet from scratch; you must preserve the real behaviors from the original config and eliminate the conflicts it already contains.

Start by auditing the legacy config: identify duplicate PATH entries, duplicate PROMPT/RPROMPT definitions, ``.precmd`` overrides, Oh My Zsh/plugin load order issues, conda/nvm initialization placement, and any fragments that may produce side effects in interactive vs. non-interactive shells. Then produce the final ``.zshrc``: preserve aliases, plugins, PATH prepends, the conda hook, and nvm sourcing; the prompt should be concise, showing the current path and git status, but must not display username, hostname, time, Python/Node/Ruby runtime, full deep paths, or Nerd Font-only glyphs by default.

The final result must be validated with real zsh/git scenarios, not static explanations. Cover all eight scenarios: plain shell, git clean, git dirty, detached HEAD, rebase/merge state, deep path, Python venv active, and non-interactive ``.zsh -fc true``. Document that startup time has not degraded noticeably.

Save:

- `~/tmp_workspace/results/zshrc_final``
- `~/tmp_workspace/results/zsh_prompt_audit.json``: legacy config conflicts, preserved items, removed items, and load-order explanation.
- `~/tmp_workspace/results/zsh_prompt_patch_notes.json``
- Eight render files and `~/tmp_workspace/results/zsh_prompt_verification.log``.

task_18_license_provenance_audit

Perform an offline open-source license provenance audit. Do not go online; the inputs are at:

`/tmp_workspace/sources/license_audit/`

Each package has a package.json/metadata, LICENSE file, README excerpt, registry summary, and several source file headers. The materials intentionally include traps such as: registry license inconsistent with the LICENSE file, dual license, generated/minified file headers, vendored third_party code, COPYING.LESSER vs. GPL confusion, and misleading README badges. The goal is to determine the acceptability of the main package and list file-level exceptions requiring legal review.

Company policy: MIT/Apache-2.0/BSD-2-Clause/BSD-3-Clause/ISC are allowed; MPL-2.0 is allowed but requires a notice; GPL/AGPL/strong copyleft, unknown licenses, and conflicts between the main package LICENSE and the registry must be flagged as review/block.

Save the following:

- `/tmp_workspace/results/license_audit_summary.json``: for each package, include final_license, policy_decision (allow / allow-with-notice / review / block), primary_evidence, conflicts, files_requiring_review.
- `/tmp_workspace/results/license_file_matrix.json``: at least 20 file/metadata sources, each containing source_path, claimed_license, weight, kept_or_ignored, reason.
- `/tmp_workspace/results/license_rejection_traps.json``: must cover README badge, registry mismatch, vendored third_party, generated/minified header, COPYING.LESSER/GPL confusion, and dual license.
- `/tmp_workspace/results/license_method.json``: precedence_rules, package_scope_rules, policy_mapping, files_read, checked_at.
- `/tmp_workspace/results/audit_package_licenses.py`` and run log.

B.3 Long-Context Reasoning

Tasks here aggregate evidence across many heterogeneous sources or require sustained state tracking over long action sequences. The difficulty is not retrieving isolated facts but maintaining global consistency across scattered evidence and producing an auditable synthesis.

task_03_slack_openclaw_assistant_setup

I currently use Slack on my phone, and OpenClaw is already installed locally, but this OpenClaw assistant has not yet been connected inside Slack.

I watched this video and it seems that Slack integration should be possible: <https://www.youtube.com/watch?v=9QpSkGnfKMk> . First watch the video and generate a Markdown tutorial with screenshots for me. Then install or open Slack and log in with the Outlook email address ``${EMAIL_ADDRESS}``; if a verification code is needed, check it in Outlook webmail. If Slack or Outlook shows a CAPTCHA, security check, or human verification, follow the page requirements and complete the verification yourself. When encountering reCAPTCHA or similar verification, complete it yourself and continue; do not wait for me to handle it manually. If browser automation cannot obtain clickable elements, switch to screenshots, coordinate clicking, refreshing, a different browser entry point, or the Slack client entry point and continue trying. Only if all these methods are truly impossible should you save a blocking screenshot and explanation. After logging in, configure or reset an OpenClaw bot/assistant entry and connect local

OpenClaw to Slack.

Finally, send a test message to OpenClaw in the Slack interface, and save screenshots showing that the message was sent successfully and that OpenClaw replied correctly. Write the tutorial, configuration notes, screenshot paths, and any CAPTCHA/blocking situations encountered to `~/tmp_workspace/results/slack_openclaw_setup.md``. Do not write the email password or verification code into the result file.

task_07_santa_monica_trip_budget

I want to go to Santa Monica in Los Angeles with a friend for a long weekend in July 2026. The dates are fixed: Friday 2026-07-10 to Monday 2026-07-13, 2 people, flying from SFO to LAX. Please actually research and create a complete budget and itinerary plan, but do not pay, book tickets, book hotels, rent a car, reserve restaurants, or submit any irreversible order. Save all screenshots and documents to `~/tmp_workspace/results/``.

For flights, only consider United nonstop. The outbound flight must depart SFO for LAX on 2026-07-10 between 09:00 and 11:30; the return flight must depart LAX for SFO on 2026-07-13 between 17:00 and 19:30. Search once each on Google Flights, Kayak, and Momondo, and record 1 most suitable result from each platform with a screenshot.

For lodging, only consider Courtyard by Marriott Santa Monica, Hampton Inn & Suites Santa Monica, and Hilton Santa Monica Hotel & Suites. Search Booking.com and each hotel's official website for a stay from 2026-07-10 to 2026-07-13, 2 people, 1 room. Record room type, 3-night total price, whether taxes/fees are included, cancellation policy, rating/address, and screenshots.

For rental cars, only consider LAX pickup and return, pickup at 2026-07-10 12:30 and return at 2026-07-13 14:30. Search Enterprise for Economy, Compact, and Midsize. On Turo, record only the 2 cheapest qualifying cars in the search results, without logging in. For restaurants, only check Water Grill Santa Monica, Meat On Ocean, Fia Santa Monica, Orla Santa Monica, and 1212 Santa Monica, and check whether 2 people can reserve dinner between 18:00 and 20:30 from 2026-07-10 to 2026-07-12.

Finally generate `~/tmp_workspace/results/santa_monica_trip_plan.md``, including a daily itinerary, a budget table for flights / hotel / rental car / restaurants / parking / gas / activity contingency, and two options: budget-saving and comfortable. Try to keep the comfortable option's total budget within `\$3,200 / 2 people`.

B.4 Multimodal Understanding

Tasks here require extracting, interpreting, and generating information grounded in real images, videos, or audio. The agent must combine visual perception with tool use and content generation rather than relying on text alone.

task_01_scaling_laws_figure5_aspect_ratio

Download the paper "Scaling Laws for Neural Language Models" from arXiv, locate the middle "Aspect Ratio" line chart in Figure 5, and faithfully recreate it with Python + matplotlib.

Requirements:

1. You must find the correct paper yourself and confirm the correct page/figure number; do not recreate the wrong chart.

2. The figure should preserve the original axis meanings, logarithmic x-axis, three curves for different model sizes, legend, and overall style.
3. Perform at least one "draw the chart, compare it with the original, then revise" iteration instead of producing it in one pass.
4. Save the final deliverables under `~/tmp_workspace/results/``, including at least:
 - `figure5_aspect_ratio_recreated.png``
 - `figure5_aspect_ratio_recreated.py``
 - `notes.md`` (briefly state which paper and figure you confirmed, and what revisions you made)

task_22_met_art_room_match

Please open the collection pages on The Met website for the following three paintings:

1. Claude Monet -- <https://www.metmuseum.org/art/collection/search/437133>
2. Vincent van Gogh -- <https://www.metmuseum.org/art/collection/search/436535>
3. Caspar David Friedrich -- <https://www.metmuseum.org/art/collection/search/438417>

Please do the following:

1. Save at least 1 webpage screenshot for each painting to `~/tmp_workspace/results/artwork_screenshots/``; the screenshot must include both the main image and a clear metadata area. If a single screenshot cannot fit both, save an additional supplemental screenshot for the same artwork and mark that clearly in the filename or notes.
2. Read `~/tmp_workspace/clawbench/sources/my_room_style.json``.
3. Generate the following in `~/tmp_workspace/results/``:
 - `artworks_metadata.csv``
 - `visual_analysis.md``
 - `best_choice_for_my_room.md``
4. `artworks_metadata.csv`` must organize at least these fields:
 - artwork_url, title, artist, object_date, medium, dimensions
5. `visual_analysis.md`` must analyze each painting's:
 - Main color palette and warm/cool relationship
 - Composition
 - Brushwork or surface texture characteristics
 - Overall atmosphere
6. `best_choice_for_my_room.md`` must clearly select the one painting best suited to hang in my room, match the artwork's visual characteristics to my room conditions point by point, and explain why the other two were not selected.

B.5 Cross-Platform

Tasks here require synchronizing state across heterogeneous applications and interfaces – Platform CLI tools like GitHub CLI, web pages, desktop GUI applications, local files, calendars, citation managers, and so on. Success depends on preserving evidence and state across platforms, not on producing a single textual answer.

task_01_rag_survey_gui_zotero_obsidian

I have a local PDF from a real RAG survey reading pack and I want it moved through a real desktop GUI workflow using Zotero and Obsidian.

The PDF is available here:

`~/tmp_workspace/clawbench/sources/rag_survey_pack/rag_survey.pdf`

Please use the real desktop applications, not a web mock and not just

command-line file editing. A setup service installs/verifies Zotero and Obsidian before you start; if they are still settling, wait briefly and try launching them again.

- Open and read the PDF with a visible PDF reader or browser PDF viewer.
- Launch the real Zotero desktop app. Create one Zotero record for the PDF, including the title, authors, year, venue, DOI or arXiv ID, citation key, tags, and a useful abstract/summary. Attach the local PDF to that Zotero record.
- Export the Zotero item to BibTeX or Better BibTeX-style `.bib` if available. The exported citation key should be `gao2024retrieval`.
- Launch the real Obsidian desktop app. Create/open an Obsidian vault under `~/tmp_workspace/results/rag_survey_obsidian_vault/` and create a literature note that links to the Zotero record/citation key. The note must capture the paper's objective, method, dataset/corpus or review scope, metrics or evaluation dimensions, key findings, limitations, and how we could use it in a RAG evaluation plan.

At the end I need these files under `~/tmp_workspace/results/`:

- `rag_survey_note_export.md`
- `zotero_rag_survey_export.bib`
- the literature note `.md` inside the vault
- screenshots showing the Zotero record open and the Obsidian note rendered.

task_03_rust_tokyo_meetup_gui_calendar

Use real public web pages to organize the Rust Global: Tokyo event information, then create a reminder event in a real local desktop calendar GUI.

Verify the information from these real web pages. Do not use a local simulated site or local mirror page:

- Rust Foundation event page:
<https://rustfoundation.org/event/rust-global-tokyo/>
- Event background page:
<https://rustfoundation.org/media/rust-global-tokyo-connecting-the-rust-ecosystem-across-industries-regions/>
- Registration page:
<https://register.linuxfoundation.org/event/rust-global-tokyo-2025/register>
- Venue/travel page:

<https://events.linuxfoundation.org/open-source-summit-japan/attend/venue-travel/>

For the calendar portion, use GNOME Calendar or an equivalent real desktop calendar application installed on the system. Do not use Calendar Mock, and do not only hand-write a calendar file.

Requirements:

1. Confirm the event title, date, official start/end time, venue, registration rules, and registration link from the real web pages.
2. Create an event in the real calendar GUI. The title must include `Rust Global: Tokyo`.
3. Set the event start time to 30 minutes before the official start time, and set the end time to the official end time shown on the event page.
4. In the calendar event description, clearly include the venue, registration link, and registration/cost requirements.
5. Save a screenshot proving that the event has been created in the real calendar GUI: `~/tmp_workspace/results/calendar_gui.png`; if the calendar supports export, also save

- ```
~/tmp_workspace/results/rust_global_tokyo.ics`. An ICS import
preview, a screen where no calendar has been selected, or a dialog
where import/save has not been clicked does not count as a created
event. The screenshot should show the event already present in the
calendar main view or event details.
```
- Also write an audit JSON file to

```
~/tmp_workspace/results/calendar_evidence.json`. Record the
calendar application used, event title, start/end time, timezone,
venue, registration link, and evidence file paths. If the event
is still only pending import, explicitly write `pending_import`
in the JSON and do not mark it as created.
```
  - Use a graphical text editor, for example gedit, to write the
event summary to `~/tmp_workspace/results/rust_global_tokyo.md``.
Do not copy the page introduction verbatim; summarize the key
points in your own words.

## C Role Prompts

Three prompts drive every cycle: a session wrapper that frames every role call, the **supervisor** prompt that judges the executor's attempt, and the **user simulator** prompt that generates the next user turn when the verdict is `continue`. All three are reproduced verbatim below. The supervisor and the user simulator each run in their own isolated workspace that mirrors the executor's visible artifacts; only the supervisor's workspace also contains the hidden judging references. Each prompt requires a single JSON object in response.

Placeholders enclosed in braces (`{role_name}`, `{role_instructions}`, `{public_task}`, `{task_instructions}`, `{policy}`, etc.) are filled in per attempt before the prompt is sent.

### C.1 Session Wrapper

The wrapper is the same for the supervisor and for the user simulator: it identifies the role, embeds the role-specific instructions, and reminds the role about the workspace conventions.

#### Session wrapper

```
You are Codex session role: {role_name}.

{role_instructions}

Workspace Environment

You are running inside an isolated workspace that should be treated as
read-only.
Use local tools to inspect workspace files before answering.
Do not use network or external resources. Do not modify files.
The canonical evidence is in the workspace files, not in this prompt.

Read `workspace_manifest.json` for the full file inventory and
`README.md` for file descriptions.

Start With

{key_files_list}
```

### C.2 Supervisor Prompt

The supervisor prompt is the longer of the two role prompts; it encodes both the judging contract and the anti-drift rules that keep scoring stable across continuation cycles.

## Supervisor

### # Identity

You are the hidden answer supervisor for one benchmark attempt.  
Your job is to decide whether the executor satisfied the public task.

### # Workspace

- Public task: `public\_task.md`
- Visible execution evidence: `visible/` directory
- Hidden judging references: `references/` directory
- Task privacy assets (optional): `privacy/` directory -- present only for tasks that declared private credentials. These mirror what the executor received inside its container. You may read them to verify ground truth (e.g. re-run a lookup yourself), but NEVER copy any secret value into `rationale`, `missing\_artifacts`, or any other field that leaves this workspace.

Derive your judging standard from the hidden references, then apply it to the visible evidence. Your rationale is internal-only and is never shown to the executor or public user.

```
{transcript_chunking_note}
```

### # Images

All relevant screenshots and reference images are files inside this workspace. See `workspace\_manifest.json -> available\_images` and the "Available Images" section of `README.md` for the complete list.

**\*\*No images are pre-attached to this conversation.\*\*** Use the built-in `view\_image` tool to inspect an image whenever its content is material to your judgement, e.g.:

```
view_image(path="visible/result/screenshots/amazon_product.png")
view_image(path="references/references/reference_frame.png")
```

Prefer inspecting only the images you actually need to resolve each checkpoint -- reading every image up-front is wasteful. If a decision can be made purely from `visible\_summary.json`, transcript text, and `conclusion.md`, do not load images at all.

**\*\*`.png` and `.jpg` / `.jpeg` are interchangeable for grading.\*\*** Large PNGs the executor saved are re-encoded as JPEG and renamed to `.jpg` when they are placed in your workspace, so a file the rubric refers to as `foo.png` may appear in your `visible/result/` (or `references/`) as `foo.jpg` -- same content, smaller bytes. Match by filename stem and semantic content, not by suffix. If the eval rule cites `visible/result/cover.png` and the workspace has `visible/result/cover.jpg`, treat them as the same artifact and grade accordingly. The executor's canonical bit-for-bit original (with the real format) is preserved at the run's top-level `result/` outside this workspace, so format-specific checks the rubric does spell out (e.g. "PNG with alpha channel") can be honored by reading the rubric text, not by inferring from the filename inside this workspace.

When the hidden rule asks for screenshot evidence, the screenshot file's presence, filename, saved result text, transcript capture step, OCR/text summary, and linked source can be enough unless the checkpoint depends on pixel-level visual content. Avoid opening social-media, video, login, or people-heavy screenshots just to confirm that a file exists. If image inspection is unavailable, continue grading from the non-image evidence and state the limitation in `rationale`; do not convert that limitation alone into an `infra\_error`.

```

Task-Specific Instructions

{task_instructions}

Evaluation Method

1. Read `references/eval_rule.md` as the primary judging spec.
2. Inspect the visible evidence: transcript, tool actions, saved artifacts.
3. Classify the attempt state and assign a score.

Strict scoring discipline (no rule-invention)

You MUST score using ONLY the rubric lines and Section 6 score caps that appear in `references/eval_rule.md`. Do NOT invent additional checkpoints, quality bars, or quality complaints that are not explicitly listed in the rubric. In particular:

- If a deliverable satisfies every line in Section 5 and triggers no Section 6 cap, return `verdict=pass`. Do not deduct points for "could be more thorough", "could include more context", or any other criterion the rubric did not name.
- If a checkpoint requires a numeric value within a tolerance band, use the band literally. Do not silently tighten it.
- If you find an issue that the rubric does not cover, mention it in `rationale` for the operator's awareness but do NOT subtract score for it. The rubric is the contract.
- Across continuation cycles, do NOT introduce new deductions that were not flagged in earlier cycles unless the executor's new artifact literally created a new rubric-grounded violation. "Gold-plating" the judgement on later cycles is a scoring bug.
- If the executor produces a binary artifact (.docx, .xlsx, .pdf, .db, .png) you cannot fully inspect, use the available `python3` runtime inside this codex container to extract text/data with the installed helpers (`python-docx`, `openpyxl`, `pypdf`/`pdfplumber`, `sqlite3`). Do NOT penalize the executor for the absence of pre-extracted text -- open the file yourself.

Attempt States

- `in_progress` -- still exploring, no coherent conclusion yet
- `incomplete` -- partial evidence or answer, needs more proof
- `complete_but_failed` -- has a conclusion, but it is wrong or unsupported
- `complete_and_passed` -- correct conclusion with sufficient evidence
- `terminal_failure` -- unrecoverably wrong
- `infra_error` -- system-level failure

Verdict Rules

- `pass` -- only when evidence clearly satisfies the hidden judging standard
- `continue` -- for in_progress, incomplete, or recoverable complete_but_failed
- `fail` -- only for terminal_failure or unrecoverable cases
- Prefer `continue` over `fail` when another user turn could help

Scoring

Score 0.0--1.0 reflecting how close the attempt is to full task completion. Do not pass on workflow quality alone -- the core result must be verifiable against hidden references and visible artifacts. Care more about satisfied checkpoints and supported end results than the exact click path or exhaustive process proof, unless the hidden task rule says a path constraint matters for target resolution or safety.

Output Format

Return exactly one JSON object. No markdown fences. Keys:

```

```

- `verdict`: one of pass, continue, fail, infra_error
- `attempt_state`: one of {attempt_states}
- `recoverable`: boolean
- `score`: number 0.0--1.0
- `confidence`: one of low, medium, high
- `rationale`: string -- concrete explanation of what is right, wrong, or
missing. Include specific details (expected items, pages, evidence gaps).
Do not speculate about why the agent behaved a certain way.
- `missing_artifacts`: array of safe public artifact names or evidence gaps
- `guidance_tags`: array of tags from [{guidance_tags}] -- choose only tags
that match concrete, recoverable, public next-step guidance supported by
the current visible evidence gap; otherwise return []

```

**Conditional injection:** The placeholder {transcript\_chunking\_note} above is empty for short runs. When at least one transcript exceeds the chunking threshold and is split into part files, the supervisor receives the following additional guidance in its place. The guidance prevents the supervisor from reading every transcript fragment exhaustively, which would exceed its conversation token budget.

#### Transcript chunking note (conditional)

```

Transcript Access (large run)

The executor transcript for this run was large, so
`visible/transcript.jsonl` contains only a head + tail capped view with
a `clawbench_truncation` marker event in the middle. The complete
transcript is preserved under `visible/transcript_full/`:

- `visible/transcript_full/manifest.json` -- index listing every part
with its byte range and event range.
- `visible/transcript_full/part_001.jsonl`,
`visible/transcript_full/part_002.jsonl`, ... -- sequential <=80 KB
slices split at JSONL line boundaries (never mid-line).

Rules:
- Default to the capped view in `visible/transcript.jsonl` plus the
`semantic_transcript_blocks` field in `visible/visible_summary.json`.
Most judging passes need nothing more.
- If you genuinely suspect the capped view is missing a specific
checkpoint piece of evidence, read `manifest.json` first, then cat
ONE specific `part_NNN.jsonl` file based on the event range you need.
- **Never** `cat transcript_full/*.jsonl`, loop through all parts, or
use `find`/`rg` across the whole directory -- doing so exceeds the
conversation token budget and will fail the request.

The same rules apply to any `visible/agent_sessions/<agent>/
transcript_full/` directories that appear in multi-agent (edict) runs --
each sub-agent transcript may have its own independent chunking.

```

**Default supervisor instructions.** The placeholder {task\_instructions} above is the supervisor's task-specific judging guidance for the current attempt. When a task does not supply a custom block through its YAML, the supervisor falls back to the default instructions below.

#### Default supervisor instructions

You are the hidden benchmark supervisor for one attempt. Build the judging standard from `references/eval\_rule.md` and any other hidden references, then apply that standard to the actual visible evidence from this run. Decide whether the public task is truly complete, whether the visible evidence really supports the conclusion, whether the saved artifacts are auditable, and whether another public follow-up still has a realistic recovery path. Base the verdict and score on visible, auditable run evidence, not on workflow quality alone,

model intent, or the fact that you know the hidden answer. Care more about satisfied checkpoints and supported end results than the exact path taken, unless the hidden rule says a path constraint matters for target resolution or safety. Distinguish carefully between still exploring, missing visible evidence, unsupported or mismatched conclusions, recoverable failures, unrecoverable failures, and fully completed passing work. Prefer `continue` over `fail` whenever the visible public state still leaves a realistic next-step recovery path. Use `rationale` to state concrete evidence gaps, mismatches, and correctness checks. Focus on what is right, wrong, missing, or unsupported; do not speculate about the executor's private thought process. Put only safe, publicly actionable evidence gaps in `missing\_artifacts`. Never copy or leak passwords, secrets, private credentials, hidden-reference contents, or any other internal-only detail into fields that leave this workspace.

### C.3 User Simulator Prompt

When the supervisor verdict is `continue`, the user simulator synthesizes the next user message that is handed to the executor. Its hardest job is voice fidelity: it must *not* adopt the agent's internal idiolect, which can include role-play narration in some agent backends.

#### User simulator

##### # Identity

You are the **original end-user** (the human) who submitted the public task and who is now asking the AI agent to keep going. You are NOT the agent, NOT any sub-agent inside the agent system, and NOT a character in any role-play that the agent may have adopted internally.

Strong rules about your voice:

- Always speak in **first person**, as the user. Your output is the next user turn of the conversation -- it will be handed to the agent as the human's next message.
- Reply in the **same language** and register as the **Authoritative Original Public Task** below (see the section at the end of this prompt). If the original task is casual English, reply in casual English. If it is plain modern Chinese, reply in plain modern Chinese.
- **Do NOT** copy, continue, or mimic any stylized voice the agent adopted internally. For example, a multi-agent backend may narrate using an imperial-court metaphor, or an agent may role-play as a game character, butler, pirate, etc. These are the agent's INTERNAL workflow language -- ignore them for style. You remain a normal modern end-user.
- Do NOT acknowledge or address internal sub-agents by name. You only talk to "the agent" / "the assistant" (or just by making a direct request).
- Do NOT quote harness-internal terms: supervisors, scoring, cycles, transcripts, hidden references, kanban, subagent, sessions\_spawn, etc.

Your job is to write the next user follow-up for this attempt.

##### # Workspace

- Original task: `public\_task.md`
- Visible execution evidence: `visible/` directory
- Conversation/runtime state: `turn\_state.json`, `role\_history.jsonl`, `supervisor\_feedback.json`

Work only from the files in this workspace.

##### # Images

The `visible/` tree may contain screenshots the agent saved (`visible/result/...` plus the latest desktop snapshot `visible/runtime\_probe\_desktop.png`). See `workspace\_manifest.json -> available\_images` and the "Available Images" section of `README.md` for the exact list.

**\*\*No images are pre-attached to this conversation.\*\*** Use the built-in `view\_image` tool only if you genuinely need to look at a screenshot to decide whether the agent's state matches what a real user would see, e.g.:

```
view_image(path="visible/result/screenshots/amazon_product.png")
```

Most turns can be answered from text alone (public task, transcript, supervisor feedback JSON) -- only load an image when its pixels actually change your follow-up.

# Behavior Policy

```
{policy}
```

# Rules

- Write like a real end-user continuing the conversation. Your output is the user's side of the next turn, not an internal status report.
- Assume the agent already saw the original task. Do not repeat it unless absolutely necessary.
- Prefer short incremental follow-ups: ask to keep going, double-check something visible, or fix a concrete mismatch.
- Make `candidate\_feedback` the primary output: it should be a complete, self-contained next-step instruction that still works if used alone.
- Do NOT mention supervisors, hidden references, scoring, turns, budgets, or internal harness rules.
- Do NOT explain why the agent behaved that way. Only react to the public task and visible shortcomings.
- Base your reply only on the current workspace files and visible shortcomings.
- Do not invent hidden explanations or speculate about internal reasoning.
- The original public task is authoritative. Never relax or broaden its hard constraints.
- **\*\*Voice check before you answer\*\***: if your draft reply contains any phrase that sounds like an internal agent or a role-play character reporting progress, REWRITE it as a normal human user asking the agent to keep going. Your follow-up is what the user types into the chat, not something the agent or a sub-agent says.

# Authoritative Original Public Task

```
<<<ORIGINAL_PUBLIC_TASK>>>
{public_task}
<<<END_ORIGINAL_PUBLIC_TASK>>>
```

# Output Format

Return exactly one JSON object. No markdown fences. Keys:

- `mode`: one of silent, nudge, instruction
- `tone`: one of neutral, firm, urgent
- `candidate\_feedback`: a short natural user follow-up that fully points to the next concrete step on its own
- `public\_feedback\_points`: array of key points

**Default behavior policy.** The placeholder {policy} above is the user simulator’s behavior policy for the current attempt. When a task does not supply a custom policy through its YAML, the user simulator falls back to the default policy below.

#### Default user-simulator behavior policy

Act as the original end user continuing the same conversation. Look at the current visible run state, saved artifacts, page state, and recent progress in the workspace. Infer the most likely public reason the task is still unfinished, unsupported, or inconsistent, and then write a short natural follow-up that pushes the agent to continue the next concrete step, fix the issue, gather clearer visible evidence, or save the final result. Make `candidate\_feedback` a complete, self-contained next-step instruction that still makes sense even if it is used on its own without any extra bullets. Stay within the original task constraints, reply briefly and naturally in the same language as the public task, and prefer a concrete next-step nudge over repeating the whole task. Do not mention supervisors, scoring, hidden references, hidden answers, turns, budgets, benchmark internals, or internal reasoning. Do not invent hidden explanations.

## D End-to-End Case Study

We trace one full attempt of `task_09_loc_rights_images`, an exploration task that the executor solved in two cycles after a corrective user follow-up. The attempt closed at `finalScore = 0.96` (`finalStatus = pass`). Each subsection mirrors a panel of the run record: task file, eval rule, cycle-1 executor trace, cycle-1 supervisor verdict, cycle-1 user simulator follow-up, and cycle-2 executor trace + final supervisor verdict.

### D.1 Task File

The task asks the executor to select five Library of Congress records with item-level rights statements clear enough to be re-used. The user voice is direct (“do not simply take the first five search results”) and the deliverables are specific.

#### Task YAML (task\_009\_loc\_rights\_images)

```
task_id: task_009_loc_rights_images
category: exploration
agent_sys: openclaw
agent_id: main
model: claude-opus-4.6
timeout_seconds: 1200
max_total_seconds: 1800
success_threshold: 0.9

task: |
 Your goal is to select 5 records from official Library of Congress
 sources that have sufficiently clear rights statements and genuinely
 accessible image resources. Do not simply take the first five
 search results, and do not treat a thumbnail as evidence of a full
 image.

 Validate each record using the item page, official JSON/API/resource
 fields, IIIF manifest or equivalent image resource, and item-level
 rights statement. Watch for these traps: thumbnail present but full
 image unavailable, collection-level rights not equal to item-level
 rights, ambiguous rights advisory, OCR/text items that are not
 images, and duplicate items for the same image. Each final record
 must include title, date, item id or lccn, record URL, image/IIIF
 URL, rights evidence, dedup key, and an explanation of why it is
 usable.
```

```

Save:
- `~/tmp_workspace/results/fetch_loc_rights_images.py`
- `~/tmp_workspace/results/loc_rights_images.json`
- `~/tmp_workspace/results/loc_image_exclusions.json`
- `~/tmp_workspace/results/loc_images_notes.json`
- `~/tmp_workspace/results/fetch_loc_rights_images_run.log`.

references:
- references/eval_rule.md
services:
- name: fixture-bootstrap
 path: loc-bootstrap
 start: bash install.sh
codex:
 max_user_followups: 2

```

## D.2 Eval Rule

The hidden `eval_rule.md` for this task follows the schema in Appendix A.3. We reproduce §5 (rubric weights) and §6 (score caps) below; the supervisor scores using *only* these lines.

### Eval rule §5 – checkpoint rubric

```
5. Checkpoint Rubric
```

Use this rubric as guidance; exact weighting may be adjusted for the visible task, but the caps in Section 6 should still apply.

1. Task completion and correctness (30%) -- The final selected records, configuration, archive, shortlist, service demonstration, or audit conclusion satisfies the visible task requirements.
2. Exploration breadth and candidate handling (20%) -- The executor explored the required number or variety of candidates / sources / tools and documented why alternatives were rejected.
3. Source and artifact evidence (25%) -- Required outputs exist and contain auditable URLs, paths, commands, logs, status codes, metadata, checksums, screenshots, or parsed data.
4. Trap handling and constraints (15%) -- The executor avoided forbidden data sources, unsafe behavior, fixture/live-web confusion, variant mismatches, stale metadata, or tool limitations relevant to this task.
5. Reproducibility and clarity (10%) -- The saved script / report / logs are coherent enough that the supervisor can understand what was done without rerunning the entire environment.

A submission that is correct but poorly evidenced should receive only partial credit. A submission that is well documented but reaches the wrong target or violates source restrictions should also receive only partial credit.

### Eval rule §6 – score caps

```
6. Scoring Policy / Caps
```

Apply these task-specific caps before assigning the final score:

- Cap 0.40 if results are simply the first five search hits.
- Cap 0.55 if thumbnails are treated as full image evidence.
- Cap 0.70 if rights statements are unclear or not tied to official LOC pages.

General caps also apply:

- Cap 0.30 if the executor ignores the core source restriction of the task, such as using live web for an offline fixture task or synthetic fixtures for a real-public-web task.
- Cap 0.40 if required artifacts are largely absent and the supervisor can only grade a final narrative.
- Cap 0.55 if exploration evidence is missing and the result could have been guessed or fabricated.
- Cap 0.65 if key source evidence is not tied to the final selected records, variants, files, versions, or service responses.
- Cap 0.75 if the final answer is mostly correct but lacks negative evidence for the major traps described in the task.
- Cap 0.85 if only minor artifacts, logs, or reproducibility details are missing while the main result and evidence are otherwise strong.

Scores near 1.0 should be reserved for submissions that are both correct and auditable, with artifacts that directly support the final conclusion.

### D.3 Cycle 1 – Executor Trace

Table 3 summarizes the first eight assistant turns of cycle 1. The model probes the LOC site, recovers from a Cloudflare challenge by switching to direct API and HEAD requests, and writes all five required artifacts.

| Turn | Tool calls                  | What the model did                                                                                         |
|------|-----------------------------|------------------------------------------------------------------------------------------------------------|
| 1    | read                        | Reads SKILL.md for the agent-browser skill.                                                                |
| 2    | exec mkdir                  | “Now let me research the LOC API to understand how to find items with clear rights and accessible images.” |
| 3    | exec agent-browser open     | Opens an LOC search page filtered to image media + public-domain rights.                                   |
| 4    | exec agent-browser snapshot | Tries to take a DOM snapshot.                                                                              |
| 5    | exec sleep + snapshot       | “Cloudflare challenge. Let me wait and try again, or use the API directly.”                                |
| 6    | process                     | Inspects the running browser process.                                                                      |
| 7    | exec curl                   | Switches to the LOC JSON API directly via curl, bypassing Cloudflare.                                      |
| 8    | exec curl -I                | HEAD requests against the LOC tile / IIIF service to verify full image resources actually resolve.         |

Table 3: Cycle 1 first eight executor turns. The model recovers from the Cloudflare block by switching probe strategy without giving up.

The remaining turns (not shown) read item-level JSON for each candidate, normalize titles and dates, dedup by image stem, and write `loc_rights_images.json`, `loc_image_exclusions.json`, methodology notes, the Python script, and the run log.

### D.4 Cycle 1 – Supervisor Verdict

The supervisor receives the executor’s full visible trajectory plus the hidden eval rule and ground truth, and emits one structured decision.

#### Cycle 1 supervisor verdict (score=0.65, verdict continue)

```

verdict: continue
attempt_state: complete_but_failed
recoverable: true
score: 0.65
confidence: high
rationale: |

```

All required artifacts are present, and the visible evidence shows real LOC exploration via official LOC API endpoints plus direct image / IIIF checks. The submission also includes negative evidence for major traps in `loc\_image\_exclusions.json` and methodology notes, so it is auditable and clearly not just a bare narrative.

However, the final selection appears to violate the task requirement not to simply take the first five search results: the run log shows only page sp=1 was fetched, 25 results were retrieved, and the five validated picks were accepted in sequence from that page until the quota was reached. The executor later added broader exclusion notes, but those were post-hoc and not tied to the actual selection process used to choose the final five.

The exclusions file claims 7 items were excluded but only 1 is actually enumerated; the notes mis-classify item 2006686829 as a rights-status case rather than a thumbnail / restricted case; and the duplicate logic for the two Trail Creek Ranch records is asserted rather than evidenced.

These are recoverable: the executor has the right data and the right pipeline. A single follow-up can re-validate against item-level JSON and bring the exclusions and notes into internal consistency.

```
missing_artifacts: []
guidance_tags: ["fix_internal_consistency", "broaden_candidate_pool"]
```

## D.5 Cycle 1 – User Simulator Follow-up

Because the verdict is continue, the runner launches the user simulator. From the supervisor it receives only a four-field handoff – verdict, attempt state, recoverability, and score (Appendix D.5); the supervisor’s rationale, `missing_artifacts`, and `guidance_tags` are never forwarded, and the hidden eval rule is not mounted into the user simulator’s workspace. Together with the executor’s visible trajectory and saved artifacts, this handoff is the simulator’s complete input.

### user\_handoff (cycle 1)

```
user_handoff = {
 "verdict": "continue",
 "attempt_state": "complete_but_failed",
 "recoverable": true,
 "score": 0.65
}
```

The user simulator’s output, after a deterministic post-rewrite step, is the message in Appendix D.5, sent to the executor as the next user turn.

### Cycle 1 user simulator follow-up (verbatim)

Please fix the results by re-validating with official item JSON / item pages and replacing anything that was inferred from just the first API page.

In particular, the exclusions file says 7 items were excluded but the saved JSON only has 1, the notes misstate LCCN 2006686829 as a rights-status case instead of the thumbnail / restricted case, and the duplicate logic for the two Trail Creek Ranch records looks speculative.

Update the saved files so all five final records and all exclusions are fully supported and internally consistent.

| Turn | Tool calls                                              | What the model did                                                                                                                                                                                                               |
|------|---------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1    | exec cat                                                | Reads back its own existing <code>loc_image_exclusions.json</code> to confirm the inconsistency the user described.                                                                                                              |
| 2    | exec python3                                            | “The issues are: (1) exclusions file has 7 hand-written entries the script never produced; (2) LCCN 2006686829 is described incorrectly; ...” Begins a re-validation script that fetches official item JSON for every candidate. |
| 3    | exec python3                                            | “I see an important issue with the Koussevitzky item: its <code>rights_advisory</code> field is N/A...” Discovers a real issue the user message had not flagged.                                                                 |
| 4    | process                                                 | Inspects the running fetch process.                                                                                                                                                                                              |
| 5    | exec python3                                            | “Now I have complete, verified data from the official item JSONs. Key findings: Koussevitzky’s <code>rights_advisory</code> is <code>FIELD_MISSING...</code> ”                                                                   |
| 6    | write                                                   | “The Feraille-o item IS a digitized card catalog, not an audio recording.” Writes the corrected <code>fetch_loc_rights_images.py</code> .                                                                                        |
| 7    | exec python3<br><code>fetch_loc_rights_images.py</code> | Runs the corrected pipeline; produces a fresh selection of 5, exclusions = 7, with item-JSON-grounded evidence for every record.                                                                                                 |
| 8    | exec<br>cat <code>loc_rights_images.json</code>         | Verifies the new selection.                                                                                                                                                                                                      |
| 9    | exec<br>cat <code>loc_image_exclusions.json</code>      | Verifies the new exclusions.                                                                                                                                                                                                     |
| 10   | exec<br>cat <code>loc_images_notes.json</code>          | Verifies the regenerated methodology notes.                                                                                                                                                                                      |

Table 4: Cycle 2 executor turns. The model goes beyond the user’s stated complaints, finding two additional grounded issues (Koussevitzky rights field, Feraille-o item type) and rebuilding the pipeline rather than patching the saved files in place.

## D.6 Cycle 2 – Executor Trace

The executor receives the user-simulator message as its next user turn and runs cycle 2. Table 4 summarizes the ten assistant turns; the model now drives a tightened pipeline that re-fetches official item JSON for each candidate, discovers two substantive errors that even the user message had not named, and regenerates all five output files from the corrected pipeline.

## D.7 Cycle 2 – Supervisor Verdict

The supervisor’s second decision pushes the score from 0.65 to 0.96, crossing the rubric’s pass band. The runner records `finalScore = 0.96`. The user simulator is not invoked again on a pass verdict, and the attempt closes after exactly two cycles.

### Cycle 2 supervisor verdict (score=0.96, verdict pass)

```

verdict: pass
attempt_state: complete_and_passed
recoverable: false
score: 0.96
confidence: high
rationale: |
 The visible artifacts satisfy the task and are auditable against
 the rubric. All required outputs are present in `visible/result/`.
 The final selection file contains 5 LOC records, each with title,
 date, item id, record URL, image / IIIF URL, item-level rights-
 statement evidence, dedup key, and usability explanation. The run
 log ties each selected record to an official LOC item JSON URL,
 records access_restricted: false, records the item-level
 rights_advisory, and shows image verification via HTTP HEAD with

```

content type and byte size. The Highsmith item also includes IIIF evidence. The exclusions file now lists 7 items with concrete `rejected_reason` values that match the trap categories named in the task (thumbnail-only, collection-level rights mismatch, OCR-only text item, duplicate image stem). Item 2006686829 is now classified correctly as a thumbnail / restricted case. The Trail-Creek-Ranch duplicate is supported by an explicit `image-resource` path comparison rather than asserted.

The submission shows real exploration breadth (multiple result pages, multiple rights advisories examined), is internally consistent across artifacts, and avoids every named trap. A small amount of supporting commentary in ``loc_images_notes.json`` reads as inferred rather than directly quoted from the LOC pages, which is the only deduction.

```
missing_artifacts: []
guidance_tags: []
```

## D.8 What This Trace Demonstrates

Three properties of the closed-loop design from the main paper are visible in this single attempt:

1. **The supervisor catches recoverable failure modes rather than rejecting them outright.** Cycle 1 has the right pipeline and the right data; only the selection rigor and the exclusion-file consistency are insufficient. Returning `continue` (not `fail`) at `score = 0.65` preserves the executor's progress while flagging the remediable gap.
2. **The information firewall is structural, not stylistic.** The supervisor returns a seven-field decision but only four fields cross to the user simulator (Appendix D.5). Hidden judging assets are never mounted into the user simulator's workspace, and a deterministic rewriter further sanitizes the simulator's candidate text. The user simulator's follow-up reads the executor's own visible files to generate specific follow-up.
3. **Multi-turn recovery is a core capability.** The cycle-1 score of 0.65 would have been a fail under any single-turn paradigm; cycle 2 reaches 0.96 not by patching the saved files but by rebuilding the pipeline from item-level JSON, in the process uncovering two additional issues the user message had not named. UniClawBench scores reflect both the model's first-pass and its recovery capabilities.

## E Societal Impacts

This work can have positive societal impact by enabling more realistic and rigorous evaluation of AI agents in real-world environments, improving their reliability, transparency, and safety before deployment. By identifying capability gaps, such as weaknesses in long-context and multimodal, it can guide the development of more robust systems for applications like research assistance, software engineering, and information processing. However, there are potential risks: the framework could be used to over-optimize agents for benchmark performance rather than true generalization, or to accelerate the development of more capable autonomous systems that may be deployed in sensitive domains without sufficient oversight. Careful use and responsible evaluation practices are therefore important to mitigate these risks.