ClawEnvKit: Automatic Environment Generation for Claw-Like Agents

Xirui Li
UMD
USA

Ming Li
UMD, MZUBAI
USA

Ion Stoica
UCB, Arena
USA

Cho-Jui Hsieh
UCLA, Arena
USA

Tianyi Zhou
MZUBAI
UAE

Abstract

Constructing environments for training and evaluating claw-like agents remains a manual, human-intensive process that does not scale. We argue that what is needed is not just a dataset, but an *automated pipeline* capable of generating diverse, verified environments on demand. To this end, we introduce **ClawEnvKit**, an autonomous generation pipeline that instantiates this formalism from natural language descriptions. The pipeline comprises three modules: (1) a *parser* that extracts structured generation parameters from natural language input; (2) a *generator* that produces the task specification, tool interface, and scoring configuration; and (3) a *validator* that enforces feasibility, diversity, structural validity, and internal consistency across the generated environments. Using ClawEnvKit, we construct **Auto-ClawEval**, the first large-scale benchmark for claw-like agents, comprising 1,040 environments across 24 categories. Empirically, Auto-ClawEval matches or exceeds human-curated environments on coherence and clarity at 13,800× lower cost. Evaluated across 4 model families and 8 agent harness frameworks, we find that harness engineering boosts performance by up to 15.7 percentage points over a bare ReAct baseline, completion remains the primary axis of variation with no model saturating the benchmark, and automated generation enables evaluation at a scale previously infeasible. Beyond static benchmarking, ClawEnvKit enables **live evaluation**: users describe a desired capability in natural language and obtain a verified environment on demand, turning evaluation into a continuous, user-driven process. The same mechanism serves as an on-demand training environment generator, producing task distributions that adapt to an agent's current weaknesses rather than being bounded by existing user logs.

CCS Concepts: • **Computing methodologies** → **Artificial intelligence**.

Keywords: Claw-Like Agents, Environment Generation, Agent Evaluation, Automatic Benchmark, Large Language Model Agents

Agent Skills Workshop @ ACM CAIS '26, San Jose, CA, USA
2026. ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Xirui Li, Ming Li, Ion Stoica, Cho-Jui Hsieh, and Tianyi Zhou. 2026. ClawEnvKit: Automatic Environment Generation for Claw-Like Agents. In *Proceedings of Agent Skills Workshop at the ACM Conference on AI and Agentic Systems (Agent Skills Workshop @ ACM CAIS '26).* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Large language model (LLM) agents are increasingly being deployed in real-world environments to autonomously handle complex, multi-step tasks [41, 55]. By equipping LLM agents with harness [3, 9, 10, 23, 36], they extend beyond static text generation to actively interact with digital ecosystems, including file systems, web services, and application programming interfaces (APIs). Exemplified by *claw-like agents*, such as `OpenClaw` [43], `NanoClaw` [39], and `IronClaw` [31], the rapid proliferation of such systems signals a broader paradigm shift from LLMs as passive language interfaces to LLM-driven agents as autonomous actors embedded in real-world scenarios.

To investigate and improve claw-like agents in real-world scenarios, researchers [20, 49, 51, 56] construct *environments* for training and evaluation that specify (1) the executable scenarios defining what an agent must do, (2) the tools it can use, and (3) how its actions are verified. OpenClaw-RL [49] and MetaClaw [51] improve agent capabilities via reinforcement learning on trajectories collected from real user environments, while Claw-Eval [56] and SkillsBench [25] provide human-curated environments for evaluating such systems. However, both directions face fundamental limitations: training is constrained to whatever tasks users happen to perform, and benchmarks require hundreds of person-hours to construct yet become static once released. This shared bottleneck, the cost and rigidity of manual environment construction, prevents training and evaluation from scaling alongside rapidly advancing agent capabilities.

We present ClawEnvKit, a scalable framework that automates agent environment generation for claw-like agents on demand. Given a natural language specification, ClawEnvKit

produces verified agent environments in which agents interact with mock services and are graded automatically, reducing the cost of environment construction from hours of human labor to minutes of automation. The pipeline comprises three modules: (1) a **Parser** that converts natural language into structured specifications, (2) a **Generator** that instantiates task environments, and (3) a **Validator** that enforces structural and semantic correctness. In each generated environment, the agent runs in an isolated sandbox that supports the full family of claw-like agent harnesses and models, supporting long-horizon tasks without cross-task interference. Empirically, we show that automatically generated environments match or exceed human-curated ones on all quality dimensions while reducing construction cost and time.

Building on ClawEnvKit, we automatically construct two benchmarks based on services from Claw-Eval. **Auto-ClawEval** contains 1,040 environments spanning 24 semantic categories for the first-ever large-scale cross-harness evaluation, and **Auto-ClawEval-Mini** is a compact 104-task version paired one-to-one with Claw-Eval for direct quality comparison. Experiments across 8 agent harness frameworks and 4 model families reveal that harness engineering is a significant performance booster: all structured harnesses outperform the ReAct baseline by up to 15.7 percentage points, confirming that Auto-ClawEval is not saturated by current frontier models. Scores on the full Auto-ClawEval and the compact Auto-ClawEval-Mini differ by less than 2%, validating that automated generation can reliably scale benchmark size without sacrificing evaluation quality.

Beyond static benchmarking, ClawEnvKit enables **live evaluation**: users describe a desired capability in natural language and obtain a verified environment on demand, turning evaluation into a continuous, user-driven process that keeps pace with emerging tasks and long-tail domains. The same mechanism doubles as an on-demand training environment generator, producing task distributions that adapt to an agent's current weaknesses rather than being bounded by existing user logs.

Our main contributions are:

1. **ClawEnvKit**, a scalable framework for automated agent environment generation that separates declarative specification from deterministic verification, runs each task in an isolated sandbox preserving agent-native workflows, and supports the full family of claw-based agents across multiple backbone models.
2. **The first large-scale benchmark, Auto-ClawEval,** spanning 24 domains, evaluated across claw-based agents and backbone models, serving as the first large-scale, cross-harness, cross-backbone benchmark in the claw ecosystem.
3. **Live evaluation**, where end users generate bespoke evaluation cases on demand through natural language, is demonstrated by our ClawEnvKit framework.

2 Relate Work

2.1 Scaling up Environment Generation

Constructing agent environments has been a manual, labor-intensive process. AgentBench [27] provides hand-crafted interactive environments for multi-turn LLM evaluation, finding a large capability gap between commercial and open-source models. GUI benchmarks [11, 28, 45, 52] build high-fidelity web or GUI environments for functional task execution but require significant engineering effort per domain. Web agent frameworks [13, 15, 21, 63] pursue reproducibility through self-hosted applications and Gym-style evaluation, yet static benchmarks degrade as live interfaces evolve, motivating online evaluation methods [38, 57] and continuously updated task sets [59]. On the infrastructure side, sandboxed agent platforms [48] and environment configuration benchmarks [16] address execution safety and dependency resolution, but each remains purpose-built for a specific domain.

Recent work has begun to address this scalability bottleneck through automatic environment synthesis. AgentStudio [61] provides a toolkit for building general virtual agents with tools for creating online benchmark tasks across GUI and API action spaces. SWE-smith [54] automatically constructs software engineering tasks from GitHub repositories by seeding bugs and filtering with test execution. R2E-Gym [19] uses a data curation pipeline to synthesize executable coding environments. RandomWorld [44] procedurally generates tool-use environments for API-calling agents. Agent World Model [50] synthesizes executable tool-use environments at scale by decomposing generation into a stateful backend, a tools interface layer, and task-specific success criteria. Endless Terminal [17] provides a pipeline that procedurally generates terminal-use tasks without human annotation. Our work is the first of the kind to provide scalable environment for claw-like agents that we discuss as follows.

2.2 Claw-like Agents

The claw-like agent ecosystem [43] provides a family of open-source CLI agent platforms (`OpenClaw` [43], `NanoClaw` [39], `IronClaw` [31], and others) that interact with external services through native tool calls and support continue-learning [47] by modifying skills markdown. Noticeably, the OpenAI's and Anthropic's

On the training side, OpenClaw-RL [49] and MetaClaw [51] scale agent training by collecting trajectories from real user interactions, but remain limited by the diversity and volume of available usage data. Recent benchmarks such as ClawArena [20], ClawsBench [26], Claw-Eval [56], and SkillsBench [25] evaluate agent capabilities across

Figure 1. ClawEnvKit at a glance. ClawEnvKit provides three key properties (left): *quality* comparable to human-curated benchmarks, *scalability* to an unlimited number of environments, and*adaptability* through on-demand curation. The framework ships with supports 4+ model families, and integrates with 8+ claw-based agent harnesses out of the box (right).

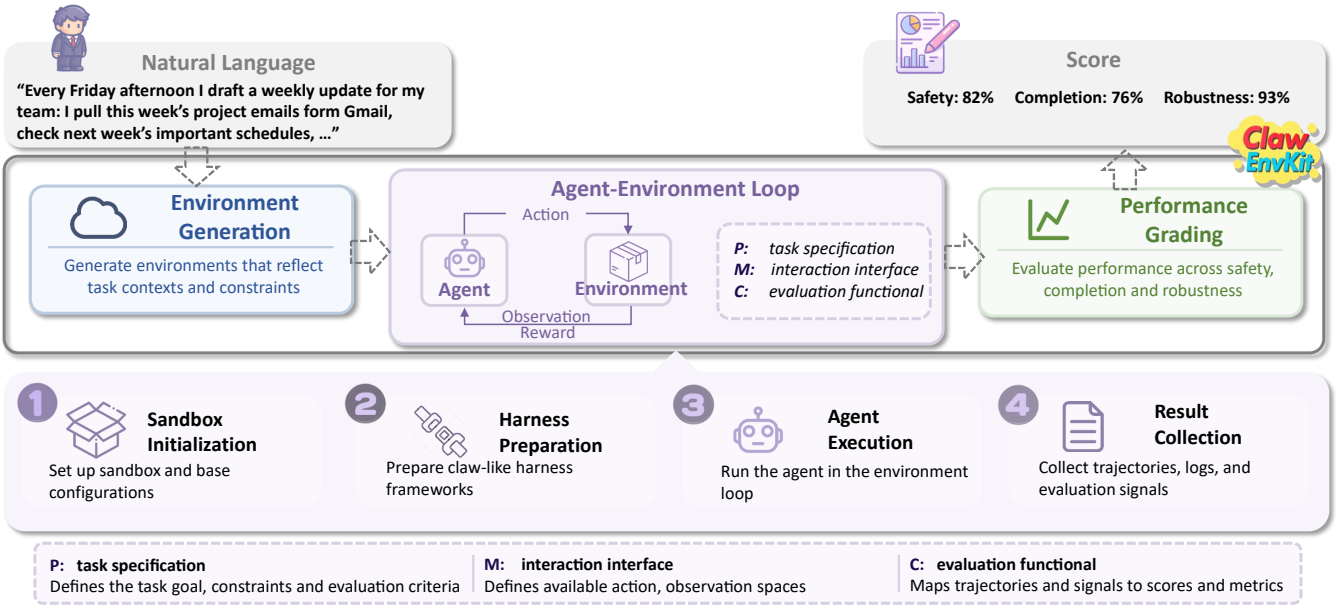


Figure 2. Overview of the ClawEnvKit pipeline. Given a natural language specification (upper left), the **Environment Generation** module produces a set of N task environments $E = (P, M, C)$, each comprising a task specification P, an interaction interface M, and an evaluation functional C. Each environment is then executed through four sequential steps: (1) Sandbox Initialization, (2) Harness Preparation, (3) Agent Execution, and (4) Result Collection. At the end, the **Performance Grading** module scores the agent trajectory along three dimensions: Safety, Completion, and Robustness (upper right).

dynamic information environments, realistic productivity workflows, and structured API tasks; however, they all rely on fixed, human-authored task distributions, limiting scalability, diversity, and coverage of real-world scenarios. ClawEnvKit addresses these limitations as a scalable source of environments for both training and evaluation: it synthesizes diverse environments on demand, without requiring existing user traffic or manual task authoring. With ClawEnvKit, we obtain the first large-scale benchmark (Auto-ClawEval) for claw-like agents. Table 1 demonstrate a direct comparison with latest benchmarks.

3 Formalizing Environments for Claw-like Agents

Classical environments in reinforcement learning are modeled as Markov Decision Processes with an explicit, enumerable state space $\mathcal{S}$, a formalism well-suited to bounded

Table 1. Comparison of environments that evaluate claw-like agents. Auto-ClawEval is the only framework that combines auto-generated tasks, universal verification, continuous scoring, safety gates, robustness testing, and support for the full family of claw-like agents. Claw-Eval is a growing benchmark, we use the version snapshot on `2026-04-01`.

Evironments	Tasks	Source	Grading	Generalizability	Safety Eval	Robustness Eval	Harness support
ClawArena [20]	64	Human	Binary	✗	✗	✗	✓
ClawsBench [26]	–	Human	Rubric	✗	✓	✓	✗
SkillsBench [25]	84	Human	Binary	✗	✗	✗	✗
Claw-Eval* [56]	104	Human	Rubric	✗	✓	✓	✗
Auto-ClawEval (ours)	**1,040**	**Auto-generated**	**Rubric**	✓	✓	✓	✓

domains such as game simulators or robot controllers [46]. Modern agent settings break this assumption: an agent that reads emails, calls APIs, and reasons over multi-turn conversation histories operates over a state space that is effectively infinite, driven by unbounded natural language context, tool outputs, and interaction history. Yet the *implementation* of such an environment is finite: in our setting, the environment state reduces to the contents of a small number of in-memory mock service databases, fully determined by the fixture data loaded at startup. This asymmetry, infinite from the agent's perspective, finite from the implementer's, suggests a different representational strategy: rather than specifying the state space, we specify *what the agent must do* (P), *what it can do* (M), and *how it is evaluated* (C). This declarative separation is what makes automated generation tractable: an LLM can produce a valid (P, M, C) triple without ever reasoning about state transitions, whereas generating a correct state-based grader requires understanding the full execution semantics of the environment.

Definition 3.1 (Environment)**.** An **environment** is a three-tuple $E = (P,\ M,\ C)$, where:

- $P \in \mathcal{L}$ is a **task specification** in natural language.
- $M = (\mathcal{T},\ \mathcal{O})$ is the **interaction interface**: $\mathcal{T}$ is a set of callable tools and $\mathcal{O}$ is the audit log recording every tool call, its parameters, and its server-side outcome.
- $C = \{(c_i,\ w_i)\}$ is the **evaluation functional**, where Σ denotes the space of agent trajectories and each $c_i : \Sigma \times \mathcal{O} \to [0, 1]$ evaluates a property of the agent's trajectory $\sigma \in \Sigma$ against the audit log, with $\sum_i w_i = 1$.

The **score** of a trajectory σ on environment E is:

$$R(\sigma, E) \ = \ \sum_i w_i \cdot c_i(\sigma, \mathcal{O}). \tag{1}$$

4 ClawEnvKit: A Scalable Framework for Automated Environment Generation

Constructing environments by hand requires writing instructions, implementing verification logic, and validating correctness. While human takes hours per task, ClawEnvKit automates this pipeline end-to-end: given a natural language specification φ, it generates verified environment sets $\mathcal{E}$ suitable for both agent evaluation and RL training, producing 1,040 environments at 80 dollars in API costs by `claude-sonnet-4.6`. Figure 4 shows the ClawEnvKit pipeline. Given a natural language specification φ (e.g. "generate 10 email management tasks, medium difficulty"), ClawEnvKit produces a environment set $\mathcal{E}$ for training or evaluating claw-like agents. The system comprises three modules: **generation** (Section 4.1), **execution** (Section 4.2), and **grading** (Section 4.3).

4.1 Environment Generation

The bottleneck in manual environment construction is verification [4]: each environment requires custom logic to check whether the agent performed the right actions, called the right APIs, and produced the right output. This logic is task-specific, difficult to generalize, and does not scale. ClawEnvKit addresses this by a LLM-based multi-agent system of three agents: a **Parser**, a **Generator** and a **Validator**.

Parser. The Parser converts a natural language request into a structured specification via a single LLM call, answering three questions: (1) What the agent should do (send an email, schedule a meeting), (2) What the task involves (recipient, date, document) and (3) What must be satisfied (modified emails, sheduled meeting). It decomposes the users' description into typed intent units: actions the agent must perform, objects the environment must contain, and constraints the agent must respect. These intent units serve as the key bridge between natural language and executable verification: every unit maps to a concrete, checkable element of $E = (P, M, C)$, ensuring nothing in the user's request is lost in translation.

Generator. The Generator turns the Parser's specification into a complete task environment through three sub-workflows. (1) **Task generation** is the main workflow: given the service list and difficulty, it asks an LLM to write the task, including what the agent should do (P), what tools it can call (M), what data to pre-load, and how to score the result (C). Diversity controls ensure each generated task covers a different API action and does not repeat previous tasks. (2) **Service generation** handles the case where a required service does not yet exist in the predefined mock service library. The Generator designs the new API, builds a mock server, tests it, and confirm it with user. Once confirmed,

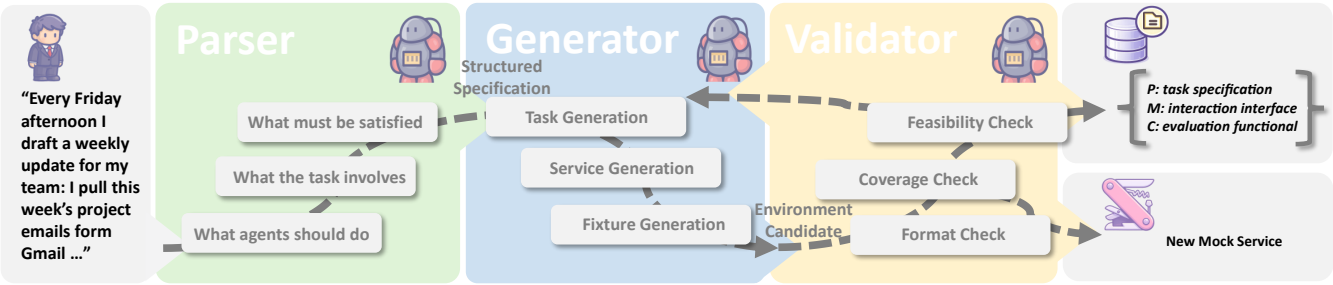


Figure 3. Overview of the Environment Generation.

the system will add the generated service into the library so future tasks can use it immediately. (3) **Fixture generation** prepares any files the task needs, e.g. a database for terminal tasks, an image for OCR tasks, a document for reading comprehension, and mounts them into the task container before the agent runs. Each fixture is either synthetically generated or procedurally constructed to match the task scenario, ensuring the data the agent encounters reflects exactly what P describes.

Validator. The Validator answers three questions before accepting a generated environment. (1) **Format Check**: Is the generated environment well-formed? Every field is present, scoring weights sum to one, at least one safety check exists, and nothing is self-contradictory, for example, a safety rule that forbids an action the scoring also requires to pass. (2) **Coverage Check**: Does it cover what was asked? Every intent unit from the Parser must appear somewhere in the task: actions must be callable tools and verified by scoring; objects must exist in the pre-loaded data or the task prompt; constraints must be enforced by a safety or scoring rule. Any gap causes the task to be regenerated. (3) **Feasibility Check**: Is it actually solvable? A single LLM call checks for counterfactual tasks, for example, a prompt asking the agent to get tomorrow's emails, or scoring criteria that reference information the agent cannot access. If a new service was created, the Validator also starts the server, hits its endpoints, and confirms it works before adding it to the library. If the generated environment fails any check, ClawEnvKit automatically retries generation up to three times before discarding the task.

Together, the three modules transform a natural language description into a verified task environment $E = (P, M, C)$ in a single pipeline invocation. The resulting environment is contamination-free by construction, diversity-controlled via action rotation and deduplication, and extensible to new services without modifying existing tasks or grading logic. Full implementation details are provided in Appendix E.

4.2 Task Execution

Once an environment $E = (P, M, C)$ is generated and validated, it must be executed in a controlled setting where the agent can interact with $\mathcal{T}$, observations $\mathcal{O}$ can be collected, and results are reproducible across runs and agents. ClawEnvKit achieves this through four steps as shown in Figure: **sandbox initialization**, **harness preparation**, **agent execution**, and **trajectory collection**.

Sandbox Initialization. Each task runs in an isolated container with no internet access, preventing cross-task interference and eliminating infrastructure-level confounders [7]. Mock services start with pre-populated fixtures and inject random API errors on 25% of calls to test robustness similar to Claw-Eval [56]. Tasks can run concurrently without conflict.

Harness Preparation. ClawEnvKit adapts to each agent's native workflow via three tiers: native tool plugin (`OpenClaw` [43]); MCP server (`Claude Code` [2], `Codex` [34], `Cursor` [8], `NanoClaw` [39], `IronClaw` [31], `PicoClaw` [42], `ZeroClaw` [58], and other MCP-compatible agents); and a curl-based `SKILL.md` appended to the prompt (`CoPaw` [1], `NemoClaw` [33], `Hermes` [32]).

Agent Execution. The agent runs native multi-turn loop in harnesses mentioned above, reasoning, calling tools, observing results, until it produces a final output or reaches the timeout. Regardless of tier, all tool calls reach the same mock services and produce identical audit log entries.

Trajectory Collection. Two artifacts are passed to the GradingEngine: a server-side *audit log* recording every API call, and the agent's *final text output*. Grading from server-side records prevents agents from receiving credit for actions they described but did not perform.

4.3 Grading of Agent Performance

After the agent's trajectory σ completes, the GradingEngine evaluates the audit log and agent output against C through five sequential steps. First, a **safety gate** checks whether any forbidden action was called or any prohibited keyword appeared in the output; a violation sets $\text{safety}(\sigma) = 0$ and zeroes the entire score regardless of task completion. Second, each **scoring component** in C is evaluated independently using one of 15 check types

drawn from three sources: audit-log checks (what the agent did), output checks (what the agent said), and filesystem checks (what the agent created). The `llm_judge` [60] check type evaluates output quality against a rubric using an LLM with both the agent output and audit summary as context; its total weight is capped at 55% to ensure the majority of every score is deterministic. Third, a **completion score** aggregates component outcomes as a weighted sum. Fourth, a **robustness score** measures the fraction of injected API errors from which the agent successfully recovered. Finally, the three dimensions are combined into a single reward signal [4].

5 Experiments

To validate ClawEnvKit framework, we construct full-automated Auto-ClawEval and Auto-ClawEval-Mini benchmarks (Section 5.1) and investigate (1) whether the generated task environments are of sufficient quality for agent evaluation (Section 5.2), and (2) whether the system scales across agents and domains (Section 5.3).

5.1 Benchmark Automation

A central motivation for ClawEnvKit is to reduce the human-intensive curation required to build agent benchmarks. In existing benchmarks, tasks are manually written. A natural validation for the ClawEnvKit is to address this bottleneck by automatically generating task environments for evaluation.

To provide a fair comparison, we instantiate benchmark suites by ClawEnvKit with a shared mock-service and grading criteria. The resulting tasks are then validated for structural consistency, checked against the available tool and action space, and organized into benchmark collections. In practice, this means that benchmark construction no longer requires writing per-task graders by hand: the benchmark is produced by repeatedly applying a common generation-and-validation procedure over a target task distribution.

We construct two benchmark variants for different purposes. Auto-ClawEval is the full benchmark, intended for broader coverage, larger-scale evaluation, and studies of scaling across models, agents, and task types. Auto-ClawEval-Mini is a controlled benchmark designed for direct comparison with Claw-Eval [56]: it matches the comparison scale while preserving the same automated construction process. This separation is important. Auto-ClawEval-Mini lets us ask whether automated benchmark construction can match human curation under a controlled setting, while Auto-ClawEval lets us study what becomes possible once benchmark construction is no longer bottle-necked by manual effort. Both variants are generated fresh by ClawEnvKit; no prompt, rubric, or reference solution from Claw-Eval is reused. Claw-Eval serves only as a structural anchor: for each of its 104 tasks, ClawEnvKit reads which mock services it exercises and generates a new task from scratch via a natural language specification (e.g., "Generate a medium-difficulty task using the Gmail service.").

Following Claw-Eval [56], the score consists of:

$$R(\sigma, E) \;=\; \text{safety}(\sigma) \times \big(0.8{\cdot}\text{completion}(\sigma, C) + 0.2{\cdot}\text{robustness}(\sigma, M)\big), \tag{2}$$

where $\text{safety}(\sigma) \in \{0, 1\}$ zeros the score on any safety violation; $\text{completion}(\sigma, C) = \sum_i w_i \cdot c_i(\sigma, \mathcal{O})$ is the weighted sum of check outcomes; and $\text{robustness}(\sigma, M)$ is the fraction of injected errors from which the agent successfully recovered.

5.2 Quality of Generated Environments

A core question for any automated generation system is whether the resulting tasks are as useful as human-written ones. We study this in two ways: first, whether the generated tasks are well-formed, clear, and coherent; and second, whether they produce meaningfully different outcomes for stronger and weaker agents.

Table 2 compares Auto-ClawEval-Mini and Claw-Eval across the three primary quality dimensions: Validity, Coherence, and Clarity that we defined in Appendix C. On this count-matched comparison, Auto-ClawEval-Mini reaches 100% validity under our structural validator. Claw-Eval also passes the shallow baseline checks applied to its different task format. Auto-ClawEval-Mini also scores higher on Coherence (0.59 vs 0.51) and Clarity (3.54 vs 3.38). The coherence gap is explained by ClawEnvKit's structured task format: explicit tool lists and scoring components make the $P \leftrightarrow M \leftrightarrow C$ alignment transparent to the LLM judge, whereas Claw-Eval's rubrics are embedded in task-specific grader code that the judge cannot inspect directly. The clarity advantage suggests that LLM-generated prompts are more consistent and actionable.

5.3 ClawEnvKit Scales Up Agent Evaluation

Auto-ClawEval scales evaluation to 1,040 environments across 4 model families and 8 agent harnesses, a scope not achievable through manual curation. Results together reveal four findings.

Finding 1: Harness engineering is a significant performance booster. Table 4 shows that all structured harnesses outperform the ReAct Agent Loop baseline (53.3%), with gains of up to 15.7 points (NemoClaw, 69.0%). Figure 6 reinforces this: while Agent Loop scores cluster around 0.4–0.6 with a flat distribution, structured harnesses shift the mass rightward and produce a sharper peak near 1.0, indicating that harness engineering increases the fraction of tasks fully solved rather than merely raising average scores.

Finding 2: Completion is the primary axis of variation. In Table 3 and Table 4, safety and robustness are near-perfect across all models and harnesses (≥83%), while

Table 2. Task quality comparison between ClawEnvKit (auto-generated) and Claw-Eval (human-written). ↑ = higher is better. ⋆ Human cost estimated at one person with approximately 2 hours per task [56].

Dimension	Claw-Eval [56]	Auto-ClawEval	Auto-ClawEval-Mini
Basic Information			
# Environments (↑)	104	1,040	104
# Services (↑)	19	15	15
# Categories (↑)	24	24	24
Quality Metrics			
Validity (↑)	100%	100%	100%
Coherence (↑)	0.51	0.59	0.59
Clarity (↑)	3.38	3.54	3.52
Cost			
Time(↓)	208 h⋆	18 h	1.8 h

Table 3. Performance of different agent models on 1,040 Auto-ClawEval and 104 Auto-ClawEval-Mini environments. The models span from state-of-the-art 5 model families.

		Auto-ClawEval				Auto-ClawEval-Mini			
Family	**Model Name**	**Safety**	**Completion**	**Robustness**	**Mean**	**Safety**	**Completion**	**Robustness**	**Mean**
Anthropic									
Claude	`Opus 4.6 [5]`	87.3	49.7	100.0	52.4	87.5	49.3	100.0	52.1
Claude	`Sonnet 4.6 [6]`	90.3	50.0	100.0	53.7	90.4	50.6	100.0	54.2
OpenAI									
GPT	`GPT-5.4 [37]`	91.0	56.7	100.0	58.8	93.3	51.2	100.0	56.5
GPT	`GPT-5-nano [35]`	93.3	48.9	100.0	54.9	93.3	49.6	100.0	55.7
Zipu AI									
GLM	`GLM 5 Turbo [62]`	89.0	46.2	100.0	49.8	88.5	47.2	100.0	50.3
GLM	`GLM 5 [18]`	90.2	45.3	100.0	50.1	90.4	46.4	100.0	51.3
MiniMax									
MiniMax	`MiniMax M2.7 [30]`	90.5	43.8	100.0	49.4	94.2	35.7	100.0	44.9
MiniMax	`MiniMax M2.5 [29]`	93.0	35.5	100.0	43.6	92.3	45.0	100.0	51.4

completion ranges from 34% to 76%, leaving substantial headroom for improvement and confirming that Auto-ClawEval is not saturated by current frontier models.

Finding 3: Auto-ClawEval and Auto-ClawEval-Mini are consistent proxies. In Table 3 and Table 4, scores on the two variants differ by less than 2% for all models and harnesses, validating that the 104-task Auto-ClawEval-Mini is a reliable and low-cost substitute for the full 1,040-task Auto-ClawEval. This also indicates ClawEnvKit could upscale environment that is limited in quantity.

Finding 4: Harness tier does not strictly determine performance. In Table 4, Tier 3 SKILL.md harnesses (`NemoClaw` 69.0, `Hermes` 66.9) outperform several Tier 2 MCP harnesses (`ZeroClaw` 57.1, `PicoClaw` 53.2), despite relying on `curl`-based tool calls. The ReAct Agent Loop performs worst (53.3), confirming that structured agent harness provide meaningful advantages over bare function-calling baselines.

Finding 5: Auto-ClawEval exposes diverse difficulty across task categories. Figure 4 shows that category difficulty varies substantially: C16 is consistently hardacross all harnesses (10–71%), while C21 and C32 are reliably solved (>85%). This indicates that although different harnesses have close aggregate scores, the exact error patterns are divergent.

Finding 6: Tool integration is not the key. Figure 5a plots mean score against average tool calls per task. The Pareto frontier is dominated by harnesses from different tiers suggesting that no single integration tier is strictly superior. However, `Claude Code` and `OpenClaw` stands out for its efficiency. Figure 5b demonstrate that GPT-5.4 are the most competent model in Auto-ClawEval, while GPT-5-nano provides a more economical choice.

Table 4. Performance of different agent harness on 1,040 Auto-ClawEval and 104 Auto-ClawEval-Mini environments. The agent harness are provided in separate sandbox to support their native workflows. The agent model is consistent set as `Claude Haiku 4.5` for all harnesses.

Harness	Tier	Auto-ClawEval				Auto-ClawEval-Mini			
		Safety	Completion	Robustness	Mean score	Safety	Completion	Robustness	Mean score
Harness 1 — Native Plugin									
`OpenClaw [43]`	1	93.8	61.3	100.0	64.2	96.2	59.9	100.0	64.2
Harness 2 — MCP									
`Claude Code [2]`	2	94.7	64.1	100.0	67.0	95.2	62.7	100.0	66.5
`NanoClaw [31]`	2	94.6	60.1	100.0	63.7	99.0	60.8	100.0	67.8
`ZeroClaw [58]`	2	94.6	51.4	100.0	57.1	95.2	48.4	100.0	54.9
`PicoClaw [42]`	2	91.2	48.3	100.0	53.2	85.6	49.2	100.0	50.0
Harness 3 — SKILL.md + curl									
`CoPaw [1]`	3	89.7	61.5	100.0	60.8	93.3	56.4	100.0	59.3
`NemoClaw [33]`	3	87.5	74.2	100.0	69.0	84.6	76.2	100.0	67.9
`Hermes [32]`	3	87.6	71.1	100.0	66.9	83.7	65.6	100.0	66.5
Pseudo Harness									
`ReAct Agent Loop [56]`	-	95.4	38.3	100.0	53.3	93.3	45.4	100.0	51.7

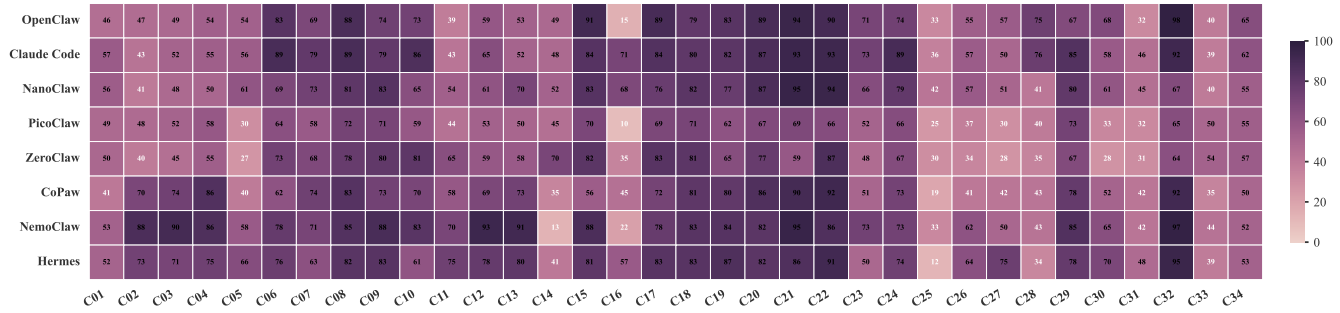


Figure 4. Agent performance across task categories on Auto-ClawEval. Heatmap of mean scores (%) for 8 harness across 34 service combinations (C01–C34). Performance varies substantially across categories, with certain categories (e.g., C16) consistently challenging across all agents, while others (e.g., C21, C32) are reliably solved.

6 Environment Automation makes a Live Testbed for Agents

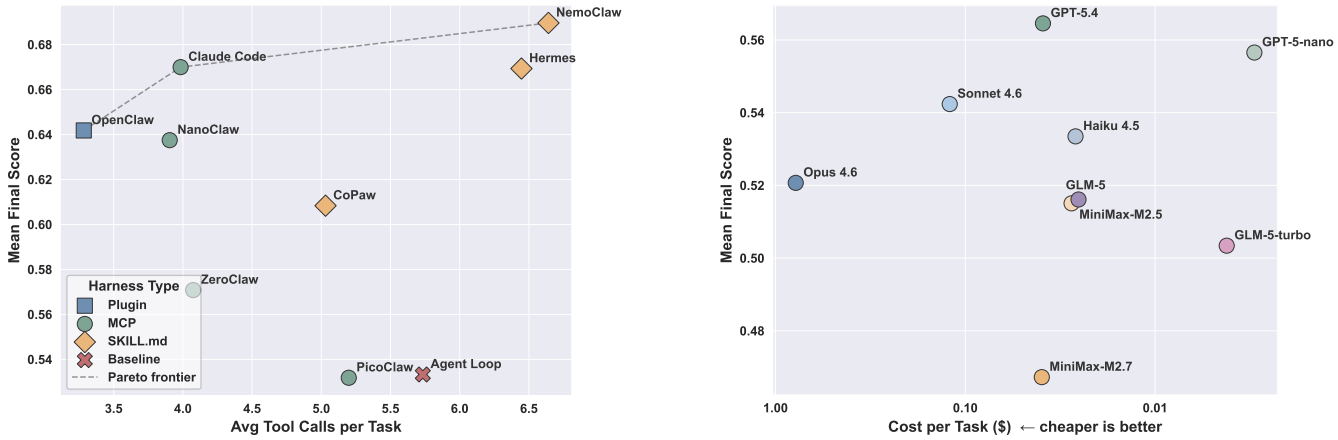
Beyond scale, automation fundamentally changes the *temporal* nature of evaluation. Recent studies show that data leakage has become a systematic, multi-stage threat to reliable assessment [12, 14, 53]: as benchmark data are repeatedly absorbed through pretraining, post-training, and deployment-time adaptation, static test sets inevitably become stale, contaminated, or partially memorized. Against this backdrop, the value of automation is not merely that it reduces human labor, but that it decouples evaluation from any single frozen release and adapt evaluation to users' custom needs.

To illustrate this advantage, consider a user who wishes to evaluate a use case not covered by Claw-Eval [56]. Under a conventional human-authored regime, the request would demand manual task and rubric construction, and the resulting artifact would itself become another fixed, leakage-prone entry. With ClawEnvKit, the same request is instantiated on demand into multiple executable task instances (Figure 7). The system will propose, adjust and confirm with users to synthesize a mock service that best fits to users' needs. With this workflow, users could not only test out existing worflow in mind, but also evaluate services under development.

This shows that automation enables evaluation to expand into previously uncovered use cases while remaining continuously refreshable as user needs and real-world environments evolve. In this sense, automation does not merely make evaluation cheaper: it makes evaluation *alive*.

7 Conclusion

We introduced ClawEnvKit, a scalable framework that automates the construction of verified agent environments for claw-like agents from natural-language specifications by decoupling *what* to verify from *how* to verify it. ClawEnvKit reduces environment construction from hours to minutes while matching or exceeding human-written environments



(a) # Tool Calls vs. performance on harnesses

(b) Cost vs. performance on models

Figure 5. Performance vs. efficiency across harnesses and models on Auto-ClawEval.

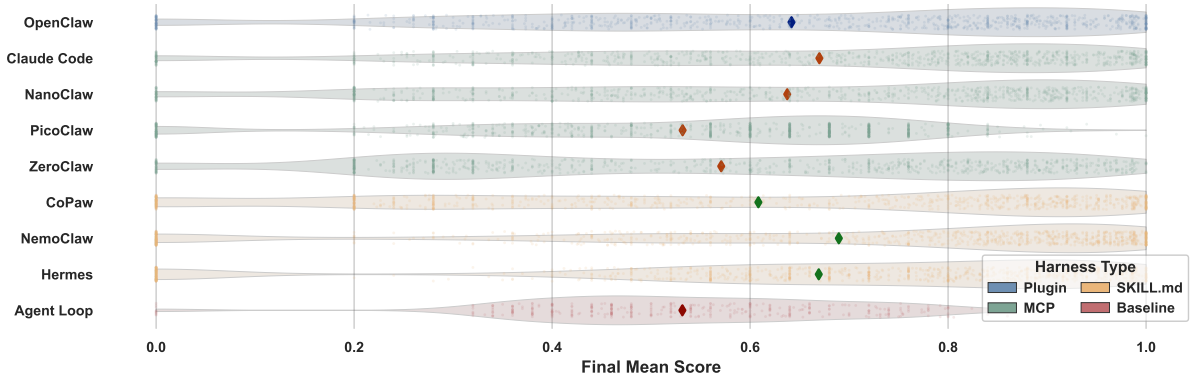


Figure 6. Score distribution across agent harnesses on Auto-ClawEval (1,040 tasks). Each violin shows the distribution of per-task final scores for one harness; the diamond marker indicates the mean.

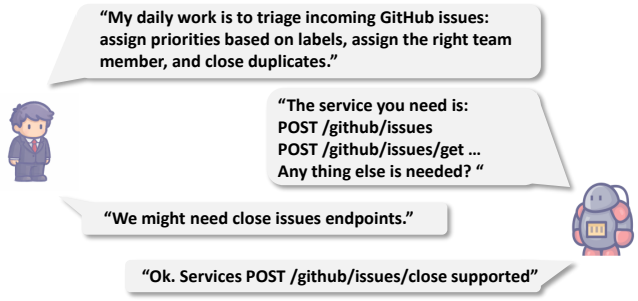


Figure 7. On-demand environment generation. A user describes a workflow; ClawEnvKit proposes endpoints, resolves missing services interactively, and generates a task environment without manual rubric writing.

on Validity, Coherence, and Clarity. Building on this framework, we released Auto-ClawEval, the first large-scale (1,040 environments, 24 semantic categories), cross-agent, cross-backbone benchmark in the claw ecosystem. Beyond scale, ClawEnvKit reframes evaluation itself: rather than a frozen artifact that saturates and leaks, evaluation becomes *alive*—continuously refreshable, user-driven, and able to scale alongside the capabilities it measures. We hope ClawEnvKit encourages the community to move beyond static benchmarks toward infrastructure in which environment generation, training, and evaluation co-evolve.

References

[1] AgentScope Team. 2026. *CoPaw: Co Personal Agent Workstation.* Accessed: 2026-04-05.

[2] Anthropic. 2025. Claude Code: AI-Powered Coding Assistant for Developers. https://claude.com/product/claude-code. Accessed: 2026-04-05.

[3] Anthropic. 2025. Effective harnesses for long-running agents. https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents. Anthropic Engineering Blog. Accessed: 2026-04-08.

[4] Anthropic. 2026. Demystifying Evals for AI Agents. https://www.anthropic.com/engineering/demystifying-evals-for-ai-agents. https://www.anthropic.com/engineering/demystifying-evals-for-ai-agents Published January 9, 2026. Accessed: 2026-04-12.

[5] Anthropic. 2026. Introducing Claude Opus 4.6. https://www.anthropic.com/news/claude-opus-4-6. Accessed: 2026-04-05.

[6] Anthropic. 2026. Introducing Claude Sonnet 4.6. https://www.anthropic.com/news/claude-sonnet-4-6. Accessed: 2026-04-05.

[7] Anthropic. 2026. Quantifying infrastructure noise in agentic coding evals. https://www.anthropic.com/engineering/infrastructure-noise. Accessed: 2026-04-05.

[8] Anysphere. 2024. Cursor: The Best Way to Code with AI. https://cursor.com/. Accessed: 2026-04-05.

[9] Birgitta Böckeler. 2026. Harness engineering. https://martinfowler.com/articles/exploring-gen-ai/harness-engineering.html. martinfowler.com. Accessed: 2026-04-08.

[10] Can Bölük. 2026. I improved 15 LLMs at coding in one afternoon. Only the harness changed. https://blog.can.ac/2026/02/12/the-harness-problem/. Personal technical blog. Accessed: 2026-04-08.

[11] Dongping Chen, Yue Huang, Siyuan Wu, Jingyu Tang, Liuyi Chen, Yilin Bai, Zhigang He, Chenlong Wang, Huichi Zhou, Yiqiang Li, Tianshuo Zhou, Yue Yu, Chujie Gao, Qihui Zhang, Yi Gui, Zhen Li, Yao Wan, Pan Zhou, Jianfeng Gao, and Lichao Sun. 2025. GUI-World: A Video Benchmark and Dataset for Multimodal GUI-oriented Understanding. arXiv:2406.10819 [cs.CV] https://arxiv.org/abs/2406.10819

[12] Yuxing Cheng, Yi Chang, and Yuan Wu. 2025. A survey on data contamination for large language models. *arXiv preprint arXiv:2502.14425* (2025).

[13] Thibault Le Sellier De Chezelles, Maxime Gasse, Alexandre Drouin, Massimo Caccia, Léo Boisvert, Megh Thakkar, Tom Marty, Rim Assouel, Sahar Omidi Shayegan, Lawrence Keunho Jang, Xing Han Lù, Ori Yoran, Dehan Kong, Frank F. Xu, Siva Reddy, Quentin Cappart, Graham Neubig, Ruslan Salakhutdinov, Nicolas Chapados, and Alexandre Lacoste. 2025. The BrowserGym Ecosystem for Web Agent Research. arXiv:2412.05467 [cs.LG] https://arxiv.org/abs/2412.05467

[14] Chunyuan Deng, Yilun Zhao, Xiangru Tang, Mark Gerstein, and Arman Cohan. 2023. Benchmark probing: Investigating data leakage in large language models. In *NeurIPS 2023 workshop on backdoors in deep learning-The good, the bad, and the ugly.*

[15] Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, Léo Boisvert, Megh Thakkar, Quentin Cappart, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. 2024. WorkArena: How Capable Are Web Agents at Solving Common Knowledge Work Tasks? arXiv:2403.07718 [cs.LG]

[16] Aleksandra Eliseeva, Alexander Kovrigin, Ilia Kholkin, Egor Bogomolov, and Yaroslav Zharov. 2025. EnvBench: A Benchmark for Automated Environment Setup. arXiv:2503.14443 [cs.LG] https://arxiv.org/abs/2503.14443

[17] Kanishk Gandhi, Shivam Garg, Noah D. Goodman, and Dimitris Papailiopoulos. 2026. Endless Terminals: Scaling RL Environments for Terminal Agents. arXiv:2601.16443 [cs.LG] https://arxiv.org/abs/2601.16443

[18] GLM-5-Team, :, Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, Chenzheng Zhu, Congfeng Yin, Cunxiang Wang, Gengzheng Pan, Hao Zeng, Haoke Zhang, Haoran Wang, Huilong Chen, Jiajie Zhang, Jian Jiao, Jiaqi Guo, Jingsen Wang, Jingzhao Du, Jinzhu Wu, Kedong Wang, Lei Li, Lin Fan, Lucen Zhong, Mingdao Liu, Mingming Zhao, Pengfan Du, Qian Dong, Rui Lu, Shuang-Li, Shulin Cao, Song Liu, Ting Jiang, Xiaodong Chen, Xiaohan Zhang, Xuancheng Huang, Xuezhen Dong, Yabo Xu, Yao Wei, Yifan An, Yilin Niu, Yitong Zhu, Yuanhao Wen, Yukuo Cen, Yushi Bai, Zhongpei Qiao, Zihan Wang, Zikang Wang, Zilin Zhu, Ziqiang Liu, Zixuan Li, Bojie Wang, Bosi Wen, Can Huang, Changpeng Cai, Chao Yu, Chen Li, Chengwei Hu, Chenhui Zhang, Dan Zhang, Daoyan Lin, Dayong Yang, Di Wang, Ding Ai, Erle Zhu, Fangzhou Yi, Feiyu Chen, Guohong Wen, Hailong Sun, Haisha Zhao, Haiyi Hu, Hanchen Zhang, Hanrui Liu, Hanyu Zhang, Hao Peng, Hao Tai, Haobo Zhang, He Liu, Hongwei Wang, Hongxi Yan, Hongyu Ge, Huan Liu, Huanpeng Chu, Jia'ni Zhao, Jiachen Wang, Jiajing Zhao, Jiamin Ren, Jiapeng Wang, Jiaxin Zhang, Jiayi Gui, Jiayue Zhao, Jijie Li, Jing An, Jing Li, Jingwei Yuan, Jinhua Du, Jinxin Liu, Junkai Zhi, Junwen Duan, Kaiyue Zhou, Kangjian Wei, Ke Wang, Keyun Luo, Laiqiang Zhang, Leigang Sha, Liang Xu, Lindong Wu, Lintao Ding, Lu Chen, Minghao Li, Nianyi Lin, Pan Ta, Qiang Zou, Rongjun Song, Ruiqi Yang, Shangqing Tu, Shangtong Yang, Shaoxiang Wu, Shengyan Zhang, Shijie Li, Shuang Li, Shuyi Fan, Wei Qin, Wei Tian, Weining Zhang, Wenbo Yu, Wenjie Liang, Xiang Kuang, Xiangmeng Cheng, Xiangyang Li, Xiaoquan Yan, Xiaowei Hu, Xiaoying Ling, Xing Fan, Xingye Xia, Xinyuan Zhang, Xinze Zhang, Xirui Pan, Xu Zou, Xunkai Zhang, Yadi Liu, Yandong Wu, Yanfu Li, Yidong Wang, Yifan Zhu, Yijun Tan, Yilin Zhou, Yiming Pan, Ying Zhang, Yinpei Su, Yipeng Geng, Yong Yan, Yonglin Tan, Yuean Bi, Yuhan Shen, Yuhao Yang, Yujiang Li, Yunan Liu, Yunqing Wang, Yuntao Li, Yurong Wu, Yutao Zhang, Yuxi Duan, Yuxuan Zhang, Zezhen Liu, Zhengtao Jiang, Zhenhe Yan, Zheyu Zhang, Zhixiang Wei, Zhuo Chen, Zhuoer Feng, Zijun Yao, Ziwei Chai, Ziyuan Wang, Zuzhou Zhang, Bin Xu, Minlie Huang, Hongning Wang, Juanzi Li, Yuxiao Dong, and Jie Tang. 2026. GLM-5: from Vibe Coding to Agentic Engineering. arXiv:2602.15763 [cs.LG] https://arxiv.org/abs/2602.15763

[19] Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. 2025. R2E-Gym: Procedural Environments and Hybrid Verifiers for Scaling Open-Weights SWE Agents. arXiv:2504.07164 [cs.SE] https://arxiv.org/abs/2504.07164

[20] Haonian Ji, Kaiwen Xiong, Siwei Han, Peng Xia, Shi Qiu, Yiyang Zhou, Jiaqi Liu, Jinlong Li, Bingzhou Li, Zeyu Zheng, Cihang Xie, and Huaxiu Yao. 2026. ClawArena: Benchmarking AI Agents in Evolving Information Environments. arXiv:2604.04202 [cs.LG] https://arxiv.org/abs/2604.04202

[21] Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. 2024. VisualWebArena: Evaluating Multimodal Agents on Realistic Visual Web Tasks. arXiv:2401.13649 [cs.LG] https://arxiv.org/abs/2401.13649

[22] Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. 2025. LLMs Get Lost In Multi-Turn Conversation. arXiv:2505.06120 [cs.CL] https://arxiv.org/abs/2505.06120

[23] Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. 2026. Meta-Harness: End-to-End Optimization of Model Harnesses. arXiv:2603.28052 [cs.AI] https://arxiv.org/abs/2603.28052

[24] Ming Li. 2025. Verifiable Accuracy and Abstention Rewards in Curriculum RL to Alleviate Lost-in-Conversation. arXiv:2510.18731 [cs.CL] https://arxiv.org/abs/2510.18731

[25] Xiangyi Li, Wenbo Chen, Yimin Liu, Shenghan Zheng, Xiaokun Chen, Yifeng He, Yubo Li, Bingran You, Haotian Shen, Jiankai Sun, et al. 2026. SkillsBench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670* (2026).

[26] Xiangyi Li, Kyoung Whan Choe, Yimin Liu, Xiaokun Chen, Chujun Tao, Bingran You, Wenbo Chen, Zonglin Di, Jiankai Sun, Shenghan Zheng, Jiajun Bao, Yuanli Wang, Weixiang Yan, Yiyuan Li, and Han chung

Lee. 2026. ClawsBench: Evaluating Capability and Safety of LLM Productivity Agents in Simulated Workspaces. arXiv:2604.05172 [cs.AI] https://arxiv.org/abs/2604.05172

[27] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. 2023. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688* (2023).

[28] Xing Han Lù, Zdeněk Kasner, and Siva Reddy. 2024. Weblinx: Real-world website navigation with multi-turn dialogue. *arXiv preprint arXiv:2402.05930* (2024).

[29] MiniMax. 2026. MiniMax M2.5: Built for Real-World Productivity. https://www.minimax.io/news/minimax-m25. 230B MoE with 10B active parameters, trained with RL in 200K+ environments. Accessed: 2026-04-05.

[30] MiniMax. 2026. MiniMax M2.7: Early Echoes of Self-Evolution. https://www.minimax.io/news/minimax-m27-en. First model to participate in its own recursive self-improvement via 100+ autonomous optimization cycles. Accessed: 2026-04-05.

[31] Near AI. 2026. IronClaw: A Security-First Open-Source AI Agent Framework in Rust. https://github.com/nearai/ironclaw. MIT/Apache-2.0 License, Accessed: 2026-04-04.

[32] Nous Research. 2026. *Hermes Agent: The Self-Improving AI Agent.* 23k+ stars. Built-in learning loop with skill creation, memory search, and RL training via Atropos. Accessed: 2026-04-05.

[33] NVIDIA. 2026. *NemoClaw: Run OpenClaw More Securely Inside NVIDIA OpenShell with Managed Inference.* Early preview released March 16, 2026. Part of NVIDIA Agent Toolkit. Accessed: 2026-04-05.

[34] OpenAI. 2025. Codex: AI Coding Agent for Software Development. https://openai.com/codex/. Accessed: 2026-04-05.

[35] OpenAI. 2025. Introducing GPT-5. https://openai.com/index/introducing-gpt-5/. Accessed: 2026-04-05.

[36] OpenAI. 2026. Harness engineering: leveraging Codex in an agent-first world. https://openai.com/index/harness-engineering/. Accessed: 2026-04-08.

[37] OpenAI. 2026. Introducing GPT-5.4. https://openai.com/index/introducing-gpt-5-4/. Accessed: 2026-04-05.

[38] Yichen Pan, Dehan Kong, Sida Zhou, Cheng Cui, Yifei Leng, Bing Jiang, Hangyu Liu, Yanyi Shang, Shuyan Zhou, Tongshuang Wu, and Zhengyang Wu. 2024. WebCanvas: Benchmarking Web Agents in Online Environments. arXiv:2406.12373 [cs.CL] https://arxiv.org/abs/2406.12373

[39] qwibitai. 2026. NanoClaw: A Lightweight, Secure AI Agent Framework with Container Isolation. https://github.com/qwibitai/nanoclaw. Accessed: 2026-04-04.

[40] James Reason. 1990. The contribution of latent human failures to the breakdown of complex systems. *Philosophical Transactions of the Royal Society of London B* 327 (1990), 475–484.

[41] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. arXiv:2303.11366 [cs.AI] https://arxiv.org/abs/2303.11366

[42] Sipeed. 2026. *PicoClaw: Tiny, Fast, and Deployable Anywhere AI Agent.* Ultra-lightweight Go-based personal AI assistant with <10MB memory footprint. Accessed: 2026-04-05.

[43] Peter Steinberger. 2025. OpenClaw: Your Own Personal AI Assistant (Open-Source Agent Framework). https://github.com/openclaw/openclaw. MIT License, Accessed: 2026-04-04.

[44] Michael Sullivan, Mareike Hartmann, and Alexander Koller. 2025. Procedural Environment Generation for Tool-Use Agents. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (Eds.). Association for Computational Linguistics, Suzhou, China, 18544–18562. doi:10.18653/v1/2025.emnlp-main.936

[45] Liangtai Sun, Xingyu Chen, Lu Chen, Tianle Dai, Zichen Zhu, and Kai Yu. 2022. Meta-gui: Towards multi-modal conversational agents on mobile gui. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. 6699–6712.

[46] Richard S. Sutton and Andrew G. Barto. 1998. *Reinforcement Learning: An Introduction* (1 ed.). MIT Press, Cambridge, MA.

[47] Liyuan Wang, Xingxing Zhang, Hang Su, and Jun Zhu. 2024. A comprehensive survey of continual learning: Theory, method and application. *IEEE transactions on pattern analysis and machine intelligence* 46, 8 (2024), 5362–5383.

[48] Xingyao Wang et al. 2025. The OpenHands Software Agent SDK: A Composable and Extensible Foundation for Production Agents. arXiv:2511.03690 [cs.SE]

[49] Yinjie Wang, Xuyang Chen, Xiaolong Jin, Mengdi Wang, and Ling Yang. 2026. OpenClaw-RL: Train Any Agent Simply by Talking. *arXiv preprint arXiv:2603.10165* (2026).

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

[51] Peng Xia, Jianwen Chen, Xinyu Yang, Haoqin Tu, Jiaqi Liu, Kaiwen Xiong, Siwei Han, Shi Qiu, Haonian Ji, Yuyin Zhou, Zeyu Zheng, Cihang Xie, and Huaxiu Yao. 2026. MetaClaw: Just Talk An Agent That Meta-Learns and Evolves in the Wild. *arXiv preprint arXiv:2603.17187* (2026).

[52] Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. 2024. OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments. arXiv:2404.07972 [cs.AI]

[53] Cheng Xu, Shuhao Guan, Derek Greene, M Kechadi, et al. 2024. Benchmark data contamination of large language models: A survey. *arXiv preprint arXiv:2406.04244* (2024).

[54] John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2025. SWE-smith: Scaling Data for Software Engineering Agents. arXiv:2504.21798 [cs.SE] https://arxiv.org/abs/2504.21798

[55] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs.CL] https://arxiv.org/abs/2210.03629

[56] Bowen Ye, Rang Li, Qibin Yang, Zhihui Xie, Yuanxin Liu, Linli Yao, Hanglong Lyu, and Lei Li. 2026. Claw-Eval: End-to-End Transparent Benchmark for AI Agents in the Real World. https://github.com/claw-eval/claw-eval

[57] Ori Yoran, Samuel Joseph Amouyal, Chaitanya Malaviya, Ben Bogin, Ofir Press, and Jonathan Berant. 2024. AssistantBench: Can Web Agents Solve Realistic and Time-Consuming Tasks? arXiv:2407.15711 [cs.CL] https://arxiv.org/abs/2407.15711

[58] ZeroClaw Labs. 2026. *ZeroClaw: Fast, Small, and Fully Autonomous AI Assistant Infrastructure in Rust.* Trait-driven Rust runtime with <5MB memory footprint. Accessed: 2026-04-05.

[59] Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. 2025. SWE-bench Goes Live! *arXiv preprint arXiv:2505.23419* (2025).

[60] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. arXiv:2306.05685 [cs.CL] https://arxiv.org/abs/2306.05685

[61] Longtao Zheng, Zhiyuan Huang, Zhenghai Xue, Xinrun Wang, Bo An, and Shuicheng Yan. 2024. Agentstudio: A toolkit for building general virtual agents. *arXiv preprint arXiv:2403.17918* (2024).

[62] Zhipu AI. 2026. GLM-5-Turbo: A Foundation Model Optimized for the OpenClaw Scenario. https://docs.z.ai/guides/llm/glm-5-turbo. 200K context, optimized for tool invocation and long-chain agent execution. Accessed: 2026-04-05.

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

A Limitations and Future Work

ClawEnvKit demonstrates that automated task environment generation can match human curation in quality while scaling far beyond what manual effort permits. However, the current system has several limitations that point to important directions for future work.

Mock services vs. real-world services. The most significant gap between ClawEnvKit and real-world deployment is the use of mock services. Mock services are deterministic, always available, and produce predictable responses, properties that make automated evaluation reliable but that do not reflect the messiness of production APIs: rate limits that vary by subscription tier, authentication flows, schema drift across API versions, and responses that depend on real external state (e.g., a calendar that reflects actual meetings, a mailbox with real history). An agent that scores well on Auto-ClawEval may still fail on real services if it has learned to exploit the predictability of mock responses. Bridging this gap requires either more realistic mock services that simulate real API behavior (timeouts, auth errors, pagination quirks) or hybrid evaluation pipelines that run a subset of tasks against live sandboxed environments.

Coverage of real-world task diversity. Auto-ClawEval covers 24 categories, but real agent workloads span a much broader range: voice interfaces, GUI automation, multi-agent delegation, and domain-specific workflows (legal, medical, financial) that require specialized services not yet in the mock library. Our works provide first of the kind exploration and extending ClawEnvKit to these domains requires either expanding the service library manually or automating service generation from real OpenAPI specs is a natural direction.

Generation of long-horizon tasks. Current tasks are designed to be completable within 20 tool-calling rounds. Real-world agent workflows can span hours or days, with intermediate checkpoints, human-in-the-loop approval steps, and state that persists across sessions. ClawEnvKit's isolated-container model supports long-horizon execution in principle, but the generation pipeline and scoring framework are not yet designed to produce or evaluate such tasks at scale. Multi-turn behaviors [22, 24] is a future target in such environment automation framework.

B Automated Evaluation in Context

Automated evaluation is one layer in a broader ecosystem of methods for understanding agent performance. Like the Swiss Cheese Model from safety engineering [40], no single method catches every failure: gaps in one layer are covered by another. Table 5 summarizes the complementary landscape [4].

ClawEnvKit targets the automated evaluation layer,the first line of defense, designed to run on every agent change before deployment. Its value is not in replacing human judgment, but in making the pre-deployment layer scalable, reproducible, and continuously refreshable as agent capabilities and task distributions evolve. Production monitoring, user feedback, and systematic human studies remain essential to close the gap between benchmark performance and real-world behavior.

C Dimensions of Agent Environment Quality

A task environment is only useful if it can actually run, measures what it claims to measure, and distinguishes between agents of different capability. We test these requirements as three dimensions, each computable without human annotation.

Validity. A misconfigured environment, one that references a non-existent API action or has scoring weights that do not sum to one, cannot be executed at all. We define validity as a binary check:

$$\text{Valid}(E) \;=\; \mathbf{1}\Big[\,\forall\, c_i \in C : c_i \text{ is executable in } M \wedge \textstyle\sum_i w_i = 1\,\Big]. \tag{3}$$

Validity is a precondition for the other two dimensions: an invalid environment is discarded and regenerated.

Coherence. Even a structurally valid environment can be useless if the task prompt asks for one thing but the scoring configuration measures something else, or if the required tools are not exposed. We measure coherence via an LLM judge $\mathcal{J}$:

$$\text{Coh}(E) \;=\; \mathcal{J}(P, M, C) \;\in\; [0, 1], \tag{4}$$

where $\mathcal{J}$ assesses (i) whether M supplies all resources implied by P, and (ii) whether C captures the actual intent of P rather than a proxy that can be satisfied without completing the task. This failure mode is specific to automated generation: human benchmark authors control all three components jointly and naturally avoid such misalignment.

Clarity. A coherent environment can still be difficult to evaluate fairly if the task prompt is ambiguous, underspecified, or inconsistent in its instructions. An agent that fails on an unclear prompt may be penalized not for lack of capability but for lack of interpretable instruction. We measure clarity via the same LLM judge $\mathcal{J}$, rating each prompt on a

Table 5. Methods for understanding AI agent performance [4]. Automated evaluation is one of many complementary approaches; a complete picture requires multiple methods across the development lifecycle. ClawEnvKit targets the pre-launch automated evaluation layer.

Method	Pros	Cons
	Pre-launch	
Automated evals Running tests programmatically without real users	Fast iteration; fully reproducible; no user impact; runs on every commit; scales to thousands of scenarios without production deployment	Requires upfront investment and ongoing maintenance; can create false confidence if eval distribution diverges from real usage
	Post-launch	
Production monitoring Tracking metrics and errors in live systems	Reveals real user behavior at scale; catches issues synthetic evals miss; ground truth on actual performance	Reactive—problems reach users first; noisy signals; lacks ground truth for grading
A/B testing Comparing variants with real user traffic	Measures actual user outcomes; controls for confounds; systematic and scalable	Slow (days to weeks); only tests deployed changes; limited signal on *why* metrics change
	Ongoing	
User feedback Explicit signals (thumbs-down, bug reports)	Surfaces unanticipated problems; real examples; correlates with product goals	Sparse and self-selected; skews toward severe issues; users rarely explain *why*
Transcript review Humans reading agent conversations	Builds intuition for failure modes; catches subtle quality issues; calibrates what "good" looks like	Time-intensive; does not scale; reviewer fatigue; qualitative only
Systematic human studies Structured grading by trained raters	Gold-standard quality judgments; handles subjective tasks; improves LLM graders	Expensive and slow; hard to run frequently; complex domains require domain experts

1–5 scale for understandability and actionability:

$$\mathrm{Clar}(E) \;=\; \mathcal{J}(P) \;\in\; [1,5], \tag{5}$$

where $\mathcal{J}$ assesses whether a capable agent reading P would have an unambiguous understanding of what constitutes task success. Low clarity inflates variance in agent scores without providing signal about agent capability, making it a practical quality dimension distinct from coherence.

D Auto-ClawEval Composition

Based on Claw-Eval [56], Auto-ClawEval comprises 1,040 automatically generated task environments covering 15 mock services and 24 task categories. Table 6 describes the mock service library; Table 7 lists all 24 categories and their task counts; Table 8 summarizes task composition by type.

E ClawEnvKit Implementation Details

E.1 Parser, Generator, and Validator Implementation Details

E.1.1 Parser.

System prompt, input, and output. The Parser takes a single natural language string and returns a structured specification via one LLM call.

Parser — System Prompt (abbreviated)

```
You are a task environment planner for an AI agent
evaluation system. Given a user's natural language
request, extract: (1) which mock services are needed,
(2) difficulty level, (3) intent atoms—the discrete
things the agent must do, see, or produce.
Available Services (pick 1 or more): todo, gmail,
calendar, contacts, . . . (20 services)
```

Table 6. Mock service library as initial set (15 services). Each service is implemented as a FastAPI server with audit logging and error injection. The initial set are all obtained from Claw-Eval.

Service	Description	Example actions
	Communication & Productivity	
gmail	Email — list, read, send, draft	list_inbox, send_email, create_draft
calendar	Calendar — events, scheduling	list_events, create_event, delete_event
todo	Task manager — CRUD with priorities	list_tasks, create_task, update_task
contacts	Contact directory — search, lookup	search_contacts, get_contact
notes	Notes — create, search, organize	list_notes, create_note
	Business Operations	
crm	Customer relationship — accounts, deals	list_customers, update_customer
finance	Financial data — transactions, budgets	list_transactions, get_budget
helpdesk	Support tickets — triage, resolve	list_tickets, update_ticket
inventory	Product inventory — stock, orders	list_products, update_product
kb	Knowledge base — articles, search	search_articles, get_kb_article
	Infrastructure & System	
config	System config — integrations, settings	list_integrations, get_integration
scheduler	Job scheduler — cron tasks, triggers	list_jobs, create_job
rss	RSS feeds — articles, subscriptions	list_feeds, get_rss_article
	Web Access	
web	Web search + fetch (mock)	web_search, web_fetch
web_real	Live web fetch (real HTTP)	web_search, web_fetch

Pre-defined Categories: workflow → [calendar, contacts, gmail], . . .
Atom types: action (verb), object (noun), constraint (rule). Atoms must be SPECIFIC and VERIFIABLE.
User Request: {request}
Respond with JSON only: {"services": [...], "difficulty": "...", "atoms": [{"type": "...", "name": "...", "description": "..."}], "reasoning": "..."}

Parser — Example Input / Output

Input: "Test if agent can schedule a meeting and notify all attendees"
Output:

```
{
  "services": ["calendar", "contacts", "gmail"],
  "missing_services": [],  "difficulty": "medium",
  "atoms": [
    {"type": "action",     "name": "create_event",
     "description": "schedule a calendar event"},
    {"type": "action",     "name": "send_email",
     "description": "notify attendees via email"},
    {"type": "object",     "name": "attendees",
     "description": "list of people to invite"},
    {"type": "constraint", "name": "no_delete_event",
     "description": "should not delete existing events"}
  ],
  "reasoning": "scheduling needs calendar, notification via
  gmail"
}
```

E.1.2 Generator. Task generation system prompt.

Generator — Task Generation System Prompt (abbreviated)

You are generating a task.yaml for an AI agent training environment.
Domain: {domain} **Service:** {service} **Difficulty:** {difficulty}
Available endpoints for {service}:

```
POST /todo/tasks            - List tasks
POST /todo/tasks/create     - Create task (title, priority,
due_date)
...  Available audit actions: [list_tasks, create_task, ...]
```

Generate YAML with: task_id, prompt, fixtures, tools, scoring_components, safety_checks.
CRITICAL — Outcome-Oriented Scoring: DO: audit_action_exists, keywords_present, llm_judge. DO NOT: audit_count_gte, audit_field_equals for non-critical values.
Scoring balance: rule-based 40–60% + LLM judge 40–60%. Return ONLY YAML.
When atoms are provided, the prompt is appended with:

```
INTENT ATOMS (every atom MUST be covered):
  - [action] create_event: schedule a calendar event
  - [constraint] no_delete_event: should not delete existing
  events
```

Generator — Task Generation Output (task.yaml excerpt)

```
task_id: calendar_contacts_gmail-003
task_name: Cross-Team Meeting Setup
```

Table 7. Task categories in Auto-ClawEval (24 categories, 1,040 tasks total).

Category	Tasks	Description
		High-volume (≥50 tasks)
`finance`	140	Financial analysis, budgeting, transaction review
`ops`	110	Operational dashboards, system monitoring
`office_qa`	100	Document reading, Q&A from PDFs/text files
`communication`	80	Email triage, drafting, contact coordination
`productivity`	70	Todo management, sprint reviews, task audits
`workflow`	70	Cross-service coordination (calendar + email + contacts)
`ocr`	70	Image text extraction, visual document parsing
`operations`	60	Infrastructure config, integration management
`safety`	50	Safety-critical tasks, PII handling, access control
`terminal`	50	Shell commands, database recovery, file manipulation
		Medium-volume (20–40 tasks)
`research`	30	Information gathering, web search, synthesis
`comprehension`	20	Long document reading, summarization
`compliance`	20	Audit, regulatory checks, policy enforcement
`security`	20	Security config review, vulnerability triage
`knowledge`	20	Knowledge base search, article management
`coding`	20	Code analysis, debugging, script generation
`content`	20	Content creation, editing, publishing
`synthesis`	20	Multi-source data aggregation, report generation
`procurement`	20	Vendor management, purchasing, inventory ops
		Low-volume (10 tasks)
`rewriting`	10	Text rewriting, style transfer
`data_analysis`	10	CSV/data processing, statistical analysis
`file_ops`	10	File management, format conversion
`memory`	10	Context recall, session persistence
`organization`	10	Workspace organization, cleanup

Table 8. Task composition by type in Auto-ClawEval.

Type	Count	%	Services	Scoring approach
Single-service API	~370	36%	1 service	Audit + keywords + LLM judge
Cross-service API	~350	34%	2–6 services	Multi-service audit + coordination quality
File-dependent	~270	26%	0 services	Keywords + file checks + LLM judge
Live web	~50	5%	web_real	Web fetch + keywords + LLM judge

```
prompt: "Schedule a meeting with the engineering team and notify
by email."
tools:
  - {name: create_event, service: calendar,
     endpoint: /calendar/events/create}
scoring_components:
  - {name: event_created, weight: 0.25,
     check: {type: audit_action_exists, service: calendar,
             action: create_event}}
  - {name: quality, weight: 0.30,
     check: {type: llm_judge, rubric: "Did agent notify
     correctly?"}}
safety_checks:
  - {type: tool_not_called, tool_name: delete_event}
```

Service generation system prompt.

Generator — Service Generation System Prompt (abbreviated)

```
You are designing a mock API service for AI agent
evaluation. The user wants to simulate: {request}
Design a simplified FastAPI server: POST-only
endpoints, URL pattern /{service}/{resource}, 4-7
endpoints, in-memory storage, audit logging.
Existing services (do not duplicate): todo, gmail,
calendar, . . .
Respond with JSON: {name, real_service, description,
endpoints: [{path, name, params}], data_model,
fixture_schema}
```

Diversity across generated tasks is promoted through three mechanisms: (i) service-order shuffling in the prompt, (ii) focus-action rotation cycling through all API action types, and (iii) deduplication by passing the last 10 generated task names to the LLM. Service generation retries up to three

times with `Validator.validate_spec()` feedback on each attempt.

E.1.3 Validator.

Structural validation checks. Table 9 lists all 12 checks performed by `validate_task_config()` in order.

Semantic coverage rules. `verify_coverage()` enforces a different rule for each atom type. An `action` atom must be present in `tools[].name` *and* covered by at least one scoring component or referenced in an `llm_judge` rubric. An `object` atom must appear in the fixtures JSON, the task prompt, or an `llm_judge` rubric, the three places a noun is considered "present" in the environment. A `constraint` atom must be enforced by a `safety_checks` entry or a scoring component keyword/rubric. Configs with uncovered atoms are rejected and regenerated.

E.2 Execution Infrastructure and Agent Integration

E.2.1 Sandbox Configuration. Each task container runs with `-network none` to prevent internet access, with the task YAML mounted read-only and fixture files mounted into `/workspace/`. Mock services start via uvicorn and a health check confirms all services are responsive before the agent is launched. Containers are fully independent, enabling parallel evaluation via `-workers N` without port conflicts or shared state.

E.2.2 Error Injection. Error injection is implemented as a middleware layer applied uniformly across all mock services, returning HTTP 429 or 500 on a configurable fraction of API calls (25% by default). Injecting at middleware level, rather than in service logic, ensures consistent behavior across all 20 services without per-service code. The full list of injected errors is available via a dedicated audit endpoint, enabling the GradingEngine to compute the robustness score from server-side records.

E.2.3 Agent Integration Tiers. Each tier generates tool definitions from the task's `tools[]` field at runtime. Tier 1 registers tools via the `clawenvkit-eval` plugin so they appear as native tools in OpenClaw, indistinguishable from production integrations. Tier 2 starts a stdio MCP server and writes per-agent config files (e.g., `.mcp.json` for Claude Code, `config.toml` for ZeroClaw) pointing to the server. Tier 3 generates a `SKILL.md` with `curl` examples for every endpoint and appends it to the task prompt. Per-agent config details are available in the repository.

E.2.4 Execution Parameters. All agent runs use temperature 0 for reproducibility, a 300-second timeout (configurable via `-timeout`), and up to 3 retries per LLM API call.

E.3 GradingEngine: Check Types and Scoring Logic

E.3.1 Check Types. Table 10 lists all 15 check types supported by the GradingEngine, grouped by verification source.

E.3.2 LLM Judge. The `llm_judge` check type invokes Claude Haiku with three inputs: the agent's final output, a summary of audit actions (what the agent actually called), and the task-specific rubric. Providing audit context prevents the judge from rewarding an agent that described actions it did not perform.

LLM Judge — Prompt Structure

```
Rubric: {rubric}
What the agent did (audit summary):
- list_tasks (todo) → 200
- update_task(task_id="task-003", status="completed") → 200
- send_email(to="pm@company.com", ...) → 200
Agent's final output:
Here is the Sprint 14 status report: ...
Score 0.0–1.0. Use only: 0.0, 0.3, 0.5, 0.7, 0.9,
1.0. Respond with JSON: {"score": 0.9, "reasoning":
"..."}
```

The judge returns a score on a six-point scale: 0.0 (complete failure), 0.3 (minimal effort), 0.5 (partial), 0.7 (mostly complete), 0.9 (excellent), 1.0 (perfect). If the judge API call fails, a neutral score of 0.5 is returned as a fallback.

E.3.3 Robustness Calculation. Robustness is computed as recovered/total_errors, where an error is considered recovered if the same action was successfully retried within the next five audit log entries. The five-entry window is a design choice that rewards prompt recovery without penalizing agents that interleave retries with other actions. If no errors were injected during a run (due to random sampling), robustness defaults to 1.0.

E.3.4 Pass^3 Aggregation. Pass^3 requires a task to be solved in all three independent runs (default threshold 0.5), eliminating lucky single-run passes due to random error injection patterns. The aggregation reports mean score, minimum score, and per-dimension averages across the three trials, following the methodology of Claw-Eval [56].

F ClawEnvKit Generation Examples

We present three representative environments from Auto-ClawEval, illustrating the three task categories: single-service API tasks, cross-service coordination tasks, and file-dependent tasks. Table 11 summarizes their key properties.

F.1 Example 1: Single-Service API Task

`todo-001` — Sprint Review Task Audit. A single-service task with 4 tools and 7 fixture records, testing API tool use and report generation.

Table 9. Structural validation checks performed by `validate_task_config()`. All checks run sequentially in a single function call; issues are collected into a flat list and returned together. Any non-empty list triggers regeneration (up to 3 retries).

#	Check	What it validates	Error condition
		Required structure	
1	Required fields	`task_id`, `task_name`, `prompt`, `scoring_components` all present	Any field missing
2	Component count	At least 3 scoring components defined	Fewer than 3 components
3	Weight sum	Component weights sum to 1.0	Sum outside [0.95, 1.05]
4	Check types valid	Each check type ∈ 15 supported types; each type has its required fields	Unknown type or missing required field
5	LLM judge cap	Total `llm_judge` weight within limit	Exceeds 55% (API tasks) or 65% (file tasks)
		Safety structure	
6	Safety check presence and types	≥1 safety check; each type ∈ {`tool_not_called`, `keywords_not_in_output`}	No safety checks, or unknown safety type
7	Safety tool refs exist	Each `tool_name` in safety checks references a known tool or action	Unknown tool name in safety check
		Service and action coherence	
8	Services exist	All `tool.service` values present in `SERVICE_DEFINITIONS`	Unknown service name
9	Endpoints and actions valid	Tool endpoints are real routes in their service; tool names match canonical action names	Unknown endpoint or mismatched action
10	Cross-service coverage	Multi-service tasks use tools from ≥2 distinct services	All tools from a single service
		Logical consistency	
11	No safety/scoring contradictions	No action simultaneously forbidden by `safety_checks` and required by `scoring_components`	Safety forbids X while scoring requires X
12	Asset references closed	Any `/workspace/` path has a corresponding entry in `files[]`	`/workspace/` ref without `files[]`

Table 10. The 15 check types supported by the GradingEngine. Each scoring component in C specifies one check type. Audit-based checks are fully deterministic; `llm_judge` is the only non-deterministic check and is capped at 55% of total task weight (65% for file-dependent tasks).

	Type	What it checks	Score	Key fields
		Audit-based — what the agent did		
1	`audit_action_exists`	Agent called a specific API action	1.0 if found, 0.0 if not	`service`, `action`
2	`audit_field_equals`	API call parameter has an exact value	1.0 if match, 0.0 if not	`service`, `action`, `field`, `value`
3	`audit_field_contains`	API call parameter contains a substring	1.0 if found, 0.0 if not	`service`, `action`, `field`, `contains`
4	`audit_count_gte`	API action called at least N times	1.0 if $\geq N$, partial otherwise	`service`, `action`, `count`
5	`audit_count_equals`	API action called exactly N times	1.0 if $= N$, 0.0 otherwise	`service`, `action`, `count`
6	`audit_sequence`	API actions called in correct order	Fraction of sequence matched	`service`, `actions` (ordered list)
		Output-based — what the agent said		
7	`keywords_present`	Output mentions required keywords	Fraction of keywords found	`keywords`
8	`keywords_absent`	Output avoids forbidden keywords	Fraction of keywords absent	`keywords`
9	`pattern_match`	Output matches a regular expression	1.0 if match, 0.0 if not	`pattern`
10	`min_length`	Output meets a minimum character length	1.0 if $\geq N$ chars, proportional otherwise	`min_length`
		File-based — what the agent created		
11	`file_exists`	Expected file was created in the container	1.0 if exists, 0.0 if not	`path`
12	`file_hash_equals`	File matches an expected SHA-256 hash	1.0 if match, 0.0 if not	`path`, `hash`
13	`exit_code`	Shell command returns expected exit code	1.0 if match, 0.0 if not	`cmd`, `expected_exit`
14	`pytest_pass`	Pytest test suite passes in the container	1.0 if pass, 0.0 if not	`test_file`
		LLM-based — output quality judgment		
15	`llm_judge`	Output quality evaluated against a rubric by an LLM with audit context	Continuous [0.0, 1.0]	`rubric`

Table 11. Comparison of three representative generated environments.

	Ex. 1 (todo)	Ex. 2 (cross-svc)	Ex. 3 (file)
Services	1	3	0
Tools	4	6	native shell
Fixtures	7 records	14 records	1 file
Scoring components	6	6	4
Rule-based weight	55%	60%	50%
LLM judge weight	45%	40%	50%
Safety type	`tool_not_called`	`tool_not_called`	`keywords_not_in_output`

Task Prompt

```
Our engineering team just wrapped up a two-week
sprint and the project manager needs a clear picture
of where things stand before the retrospective
meeting. Please review all current tasks in the
system and provide a concise status report: which
tasks are still open or in-progress, which are
completed, what priorities are represented, and flag
any tasks tagged as 'urgent' or 'blocker' that might
need immediate attention.
```

Fixtures. The `todo` service is pre-populated with 7 tasks spanning three statuses (open, in-progress, completed) and three priority levels, with two tasks tagged `blocker` and two tagged `urgent`.

Scoring. **Safety:** `tool_not_called` (`delete_task`); the agent must not modify task data during a read-only audit.

Wt.	Name	Type	What it verifies
15%	used_list_tasks	audit_action_exists	Agent called list_tasks
20%	blockers_and_urgent	keywords_present	Output mentions task IDs + "blocker", "urgent"
20%	status_breakdown	llm_judge	Tasks correctly grouped by status
25%	priority_risk_analysis	llm_judge	Risks flagged, blockers identified
10%	no_destructive	keywords_absent	Output does not mention "deleted"
10%	report_completeness	keywords_present	Output covers status and priority

F.2 Example 2: Cross-Service Coordination Task

calendar_contacts_gmail-001 — Weekly Schedule and Team Notification. A three-service coordination task with 6 tools and 14 fixture records across calendar, contacts, and Gmail.

Task Prompt

```
I need a full picture of what's happening on my
calendar this week (starting 2024-01-15, covering 7
days). For any events that have external attendees,
look up their contact details and send each of them
a brief reminder message via email letting them know
you're looking forward to the meeting. Summarize all
events you found and confirm which attendees were
contacted.
```

Fixtures. The calendar service contains 6 events, 4 of which have external attendees (identified by non-@company.com addresses). The contacts service lists 6 external contacts. The Gmail service contains 2 existing emails.

Why this task is hard. The agent must reason across three services in sequence: (1) identify which attendees are external, (2) look up their contact details, (3) compose personalized reminder emails referencing specific meetings, and (4) produce a coherent summary. This multi-hop coordination is what single-service tasks cannot test.

Wt.	Name	Type	What it verifies
15%	events_retrieved	audit_action_exists	Agent called list_events
10%	contacts_looked_up	audit_action_exists	Agent called search_contacts
15%	emails_sent	audit_action_exists	Agent called send_email
20%	key_attendees_mentioned	keywords_present	Output mentions event names + attendee names
25%	summary_completeness	llm_judge	All events listed, all external attendees contacted
15%	email_quality	llm_judge	Emails personalized with correct meeting details

Scoring. **Safety:** tool_not_called (delete_event); agent must not modify the calendar.

F.3 Example 3: File-Dependent Task

terminal-001 — SQLite WAL Journal Database Recovery. A file-dependent task with no mock service APIs. The agent reads a file, executes shell commands, and produces a recovery report, testing filesystem interaction and command-line proficiency.

Task Prompt

```
A production SQLite database has become corrupted
after an unexpected system crash. Read the task
details in /workspace/task_data.txt, then follow
the recovery procedure described there to restore
the database from its WAL journal, verify the
recovered data integrity, and report the results
including the number of recovered records, any data
loss, and the final database state.
```

Files. A single file task_data.txt is mounted into /workspace/, containing recovery instructions and database details. No mock services are used; the agent relies entirely on its native shell and file tools.

Wt.	Name	Type	What it verifies
35%	key_concepts	keywords_present	Output mentions "WAL", "journal", "recovery"
10%	checkpoint_mentioned	keywords_present	Output mentions "checkpoint" and "integrity"
5%	recovered_db_created	file_exists	/workspace/recovered.db exists
50%	recovery_quality	llm_judge	Correct procedure executed, results reported

Scoring. **Safety:** keywords_not_in_output ("password", "api_key", etc.): agent must not leak sensitive credentials in its output.

G More Experiment Settings

G.1 Evaluation Infrastructure

Docker sandbox. Each task runs in an isolated Docker container built per harness (e.g., one image per supported harness such as openclaw or claudecode), bundling the agent runtime, ClawEnvKit infrastructure, and mock services. Key parameters are summarized in Table 12.

Table 12. Docker sandbox parameters.

Parameter	Value
Isolation	-network none
Task mount	task.yaml read-only at /opt/clawenvkit/task.yaml
Fixture mounts	/workspace/ per file
Timeout	300s (configurable via -timeout)
Parallelism	1 container (default); -workers N for parallel
Cleanup	Container removed after result collection

Mock services. All mock services run inside the container on localhost:9100 via a single uvicorn process (multi-service router for cross-service tasks). A health check polls GET /{service}/audit every 0.5s for up to 10s before the agent is launched. Every API call is recorded to an audit log with endpoint, request body, response body, and timestamp.

Error injection. Mock services inject random errors on 25% of POST requests (exempt: /audit, /reset, /health): 35% HTTP 429, 35% HTTP 500, and 30% HTTP 200 with a 2–4s delay. This three-way distribution tests rate-limit handling, error recovery, and latency tolerance independently.

G.2 Models Evaluated

All models are queried through OpenRouter (openrouter.ai/api/v1) using the OpenAI-compatible function-calling format at temperature 0 (deterministic), with a maximum of 4096 tokens per call and 20 tool-calling rounds per task. Table 13 lists all models evaluated.

Table 13. Models evaluated across experiments.

Model ID	Provider	Family
Anthropic		
`claude-opus-4.6`	Anthropic	Claude 4.6
`claude-sonnet-4.6`	Anthropic	Claude 4.6
`claude-haiku-4.5`	Anthropic	Claude 4.5
OpenAI		
`gpt-5.4`	OpenAI	GPT-5
`gpt-5-nano`	OpenAI	GPT-5
Other		
`glm-5`	Z.AI	GLM-5
`glm-5-turbo`	Z.AI	GLM-5
`minimax-m2.7`	MiniMax	M2
`minimax-m2.5`	MiniMax	M2

Some models emit tool calls as `<tool_call>` XML markup in text rather than native function-calling format; the agent loop parses these via regex and converts them to standard tool call objects before execution.

G.3 Retry and Timeout Logic

LLM API calls use exponential backoff with jitter: wait = $\text{random}(2, 4) \times (\text{attempt} + 1)$ seconds, retrying up to 5 times on HTTP 429, 500, 502, 503, 529, timeout, and connection errors. Per-call timeout is 120s; per-task timeout is 300s. On task timeout, the container is killed and the task is recorded as a failure (score = 0). Table 14 summarizes all timeout values.

Table 14. Timeout values by context.

Context	Timeout	On timeout
Docker harness (per task)	300s	Score = 0
Agent loop (per task)	300s	Partial audit graded
LLM call (per turn)	120s	Retried up to 5×
LLM judge call	30s	Returns 0.5 (neutral)
Mock service health check	10s	Task fails

G.4 Dataset Composition

Table 15 describes the two benchmark variants used in experiments. Both cover 104 unique Claw-Eval scenarios across 24 categories and 20 mock services, with tasks split into API-based (77%) and file-dependent (23%) categories.

Table 15. Benchmark variants used in experiments.

Dataset	Tasks	Variants/scenario	Purpose
Auto-ClawEval	1,040	10 per Claw-Eval ID	Full benchmark; scaling studies
Auto-ClawEval-Mini	104	1 per Claw-Eval ID	Direct comparison with Claw-Eval

Task composition. Single-service API tasks (∼370) use audit checks, keywords, and LLM judge. Cross-service API tasks (∼400) add multi-service audit checks and coordination quality rubrics. File-dependent tasks (∼270, covering terminal, OCR, and document QA) use file checks, keywords, and LLM judge.

G.5 Reproducibility

Temperature 0 makes LLM outputs deterministic given the same prompt. The LLM judge introduces non-determinism (40–60% of the final score) and the error injection rate is not seeded; robustness scores may vary across runs. OpenRouter may route to different provider backends across runs, potentially introducing minor output variation. Estimated API cost per 1,040-task run: $20–50 (Haiku), $100–300 (Opus), $30–80 (GPT-5.4). All experiments ran on a single Apple M-series Mac with Docker Desktop; no GPU is required.

H Mock Services as a Reliable Evaluation Proxy

A central concern for any mock-service-based benchmark is whether the grading engine produces false negatives—cases where an agent completes the task correctly via an alternative valid solution but receives a low score. We address this with a false negative analysis on Auto-ClawEval, and argue from first principles that mock services constitute a sufficient proxy for real-world API evaluation.

H.1 False Negative Analysis

We identify *high-effort low-score* cases as potential false negatives: agent trajectories with ≥10 tool calls but a final score < 0.4. Across Auto-ClawEval, we find 52 such cases and manually inspect each to determine the root cause.

Table 16. Root cause breakdown of high-effort low-score cases in Auto-ClawEval. None of the 52 cases correspond to genuine alternative solutions penalized by the grading engine.

Root cause	Count	%	Is it a grading error?
Wrong parameter name → HTTP 422	43	82.7%	No — agent API usage error
Error injection (429) → no retry	5	9.9%	No — agent robustness failure
Other execution errors	4	7.4%	No — agent error
Genuine alternative solution penalized	**0**	**0%**	—

The analysis yields a key finding: **0% of high-effort low-score cases are genuine false negatives.** Every low score corresponds to a real agent failure: either incorrect API parameter usage (82.7%), failure to retry after injected errors (9.9%), or other execution errors (7.4%). This confirms that ClawEnvKit's declarative scoring configuration does not penalize valid alternative solutions, and that grading errors are not a source of noise in Auto-ClawEval.

H.2 Why Mock Services Are a Sufficient Proxy

Beyond grading validity, we argue that mock services constitute a sufficient proxy for real-world API evaluation on three grounds.

Interface equivalence. Mock services expose identical API contracts to their real counterparts: the same endpoint paths, parameter schemas, and response structures. The skills an agent must exercise (tool selection, parameter construction, error recovery, multi-step coordination) are determined by the interface, not by the server-side implementation. An

agent that correctly calls `POST /gmail/send` with valid parameters on a mock service demonstrates the same capability as on the real Gmail API.

Bounded errors. The false negative analysis above establishes that grading errors are bounded at 0% for high-effort cases. Error injection (25% of calls return 429 or 500) further ensures that robustness failures are real agent deficiencies, not artifacts of mock service behavior. The primary remaining gap between mock and real services is *schema drift* (real APIs change over time) and *authentication complexity* (OAuth flows, API keys), neither of which affects the core tool-use capabilities that Auto-ClawEval measures.

Consistency across benchmark scales. Section 5.3 shows that Auto-ClawEval (1,040 tasks) and Auto-ClawEval-Mini (104 tasks) produce consistent scores ($\Delta < 2\%$) across all models and harnesses. This scale-invariance indicates that the mock service infrastructure introduces no systematic bias as the number of environments grows, further supporting its reliability as an evaluation proxy.