ClawGym: A Scalable Framework for Building Effective Claw Agents

Anonymous ACL submission

Abstract

Claw-style environments support multi-step workflows over local files, tools, and persistent workspace states. However, scalable development around these environments remains constrained by the absence of a systematic framework, especially one for synthesizing verifiable training data and integrating it with agent training and diagnostic evaluation. To address this challenge, we present **ClawGym**, a scalable framework that supports the full lifecycle of Claw-style personal agent development. Concretely, we construct **ClawGym-SynData**, a diverse dataset of 13.5K filtered tasks synthesized from persona-driven intents and skill-grounded operations, paired with realistic mock workspaces and hybrid verification mechanisms. We then train a family of Claw-style models, termed **ClawGym-Agents**, through supervised fine-tuning on black-box rollout trajectories, yielding capable agents for workspace-grounded tool use, file manipulation, and persistent state interaction. To support reliable evaluation, we further construct **ClawGym-Bench**, a benchmark of 200 instances calibrated through automated filtering and human-LLM review. In summary, these components position ClawGym as a step toward scalable, verifiable, and reproducible development of Claw-style personal agents. Our resources are available at https://anonymous.4open.science/r/ClawGym-Anonymous-07D7.

1 Introduction

Recent advancements in autonomous agent frameworks, such as OpenClaw (OpenClaw, 2026b), have reshaped how AI systems interact with daily digital environments. Unlike traditional chatbots confined to isolated dialog boxes (Zhu et al., 2025), these systems are deployed within users' computer environments (Cheng et al., 2026), where they can invoke tools, manage local files, and interact with web-enabled services. This paradigm bridges machine intelligence and digital execution, enabling users to "raise" digital agents for concrete multi-step workflows (Ma et al., 2026).

Despite the capabilities of OpenClaw and similar systems (AgentScope-AI, 2026; HKUDS, 2026), autonomous agents still struggle with everyday digital tasks, even when powered by advanced large language models (Zhao et al., 2023). This is especially pronounced for less capable models, which often fail to follow multi-step instructions, select tools, or recover from errors. The difficulty stems from the distinctive nature of Claw-style tasks. Unlike static text-based reasoning tasks such as AIME (MAA, 2026), which are well specified and solvable through pure reasoning (Bai et al., 2025), Claw-style tasks are grounded in workspace states and require agents to reason over artifacts, use tools, and update workspaces. They also differ from conventional agentic benchmarks such as SWE-Bench-Verified (Chowdhury et al., 2024) and BrowseComp (Wei et al., 2025), which typically provide more structured and observable agent loops or environments (Sun et al., 2025a; Dong et al., 2025; Bai et al., 2025). In contrast, Claw-style agents often operate through opaque interfaces and must handle ambiguous instructions and unexpected states.

Recognizing these challenges, recent studies have sought to improve personal agents through specialized training algorithms (Wang et al., 2025; Xia et al., 2026b), online learning frameworks (Wang et al., 2026a), and evaluation benchmarks (Ye et al., 2026; Ding et al., 2026; Meng et al., 2026; PinchBench, 2026). However, existing work is still limited by the lack of large-scale Claw-style task data grounded in realistic workspaces. Compared with static text-based tasks and structured agent benchmarks (MAA, 2026; Chowdhury et al., 2024; Wei et al., 2025), such data is non-trivial to synthesize because it must cover personalized requirements (Patwardhan et al., 2025), support automatic verification of long-horizon workflows (Sun et al., 2025b), and provide task-specific

workspace artifacts that give agents meaningful contexts for interaction (Xie et al., 2024; Bonatti et al., 2024). These requirements leave scalable and verifiable data construction for Claw-style agent training and evaluation underexplored.

To address this gap, we propose **ClawGym**, a data-centric framework that connects task synthesis, agent training, and performance evaluation for Claw-style personal agents. We first construct **ClawGym-SynData**, a curated dataset of 13.5K executable tasks generated through persona-driven and skill-grounded synthesis. The resulting tasks are paired with task-specific workspace artifacts and hybrid verifiers that combine code-based checks with rubric-based judgment. Building on this data, we collect high-fidelity trajectories through black-box rollouts on the OpenClaw harness and train **ClawGym-Agents** to improve their ability to handle multi-step workspace-grounded workflows over tools, files, and evolving task states. For evaluation, we construct **ClawGym-Bench**, a reliable benchmark of 200 instances selected through careful human-LLM review. Experiments show that training on ClawGym-SynData yields consistent gains across model scales, while ClawGym-Bench clearly distinguishes agents with different capabilities, demonstrating the effectiveness of ClawGym for scalable data synthesis, agent training, and diagnostic evaluation.

Our contributions are threefold:

- We release ClawGym-SynData, a large-scale synthesized dataset of 13.5K executable and verifiable tasks for Claw agents. Its automated synthesis pipeline combines persona-driven intents with skill-grounded operations to construct diverse Claw-style tasks at scale.
- We collect high-quality trajectories through black-box rollouts and perform supervised fine-tuning to obtain Claw agents that better handle multi-step workspace-grounded workflows over tools, files, and evolving task states.
- We construct a benchmark of 200 instances across six categories for evaluating Claw agents, with reliability ensured through human-LLM review. The benchmark reveals clear capability gaps across model scales.

2 Related Work

Evaluating Autonomous Agents. With the rapid progress of LLMs and harnesses (Tang et al., 2026), autonomous agents have increasingly moved beyond static text generation toward solving complex, long-horizon, and personalized tasks (Ding et al., 2026). Unlike conventional text-only reasoning benchmarks (MAA, 2026), general-purpose agents must operate in interactive digital environments, where they can invoke tools, manipulate files, browse web resources, and update persistent workspace states (Li et al., 2025). Recent benchmarks have evaluated agent capabilities across office workflows (Chen et al., 2026), professional domains (Yang et al., 2026), and personalized Claw-style tasks (PinchBench, 2026; Ye et al., 2026). However, many existing evaluations are costly to scale and are primarily designed for model assessment, rather than for constructing reusable executable tasks with workspace resources and verifiers that can also support agent training.

Training General Agents. To train general agents for diverse, personalized, and long-horizon tasks, prior work has explored several directions. These include online learning from user interaction traces (Wang et al., 2026a, 2025), where human feedback naturally provides quality signals for reinforcement learning, synthetic task generation and environment scaling to broaden training coverage (Song et al., 2026b; Wang et al., 2026b; Dong et al., 2026), and meta-learning or skill reuse mechanisms to improve cross-task adaptation (Xia et al., 2026b,a). However, these studies still provide limited support for constructing executable Claw-style task instances that jointly specify realistic workspace resources and verifiable success conditions, especially for personalized environments such as OpenClaw (OpenClaw, 2026b).

3 ClawGym-SynData: Scaling Effective Task Synthesis for Claw Agents

As shown in Figure 1, ClawGym-SynData follows a four-stage pipeline for scalable task synthesis, including 1) Task Generation, 2) Resource Preparation, 3) Verification Design, and 4) Automated Quality Assessment. This pipeline supports large-scale data construction while preserving diversity, execution realism, and verifiable quality.

3.1 Task Generation

3.1.1 Persona-Driven Top-Down Synthesis

Persona-driven top-down synthesis generates Claw-style tasks from high-level user contexts. It first constructs a task seed that specifies the intended

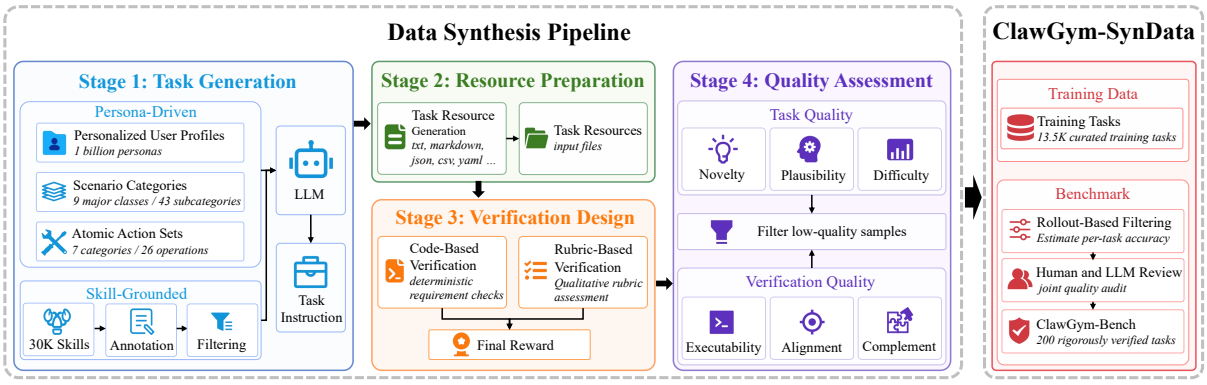


Figure 1: Overview of the ClawGym-SynData pipeline, which generates tasks from persona-driven and skill-grounded sources, prepares task resources, designs hybrid verification, filters samples through quality assessment, and constructs training and benchmark data.

user, scenario, and required operations, and then expands the seed into a multi-step instruction.

Seed Formulation. Each task seed is defined as $z = (u, c, \mathcal{G})$, where $u \in \mathcal{U}$ is a user persona, $c \in \mathcal{C}$ is a scenario category, and $\mathcal{G} = \{g_1, \ldots, g_n\}$ is a set of basic operations. The scenario taxonomy contains 9 major classes and 43 subcategories, and the operation taxonomy contains 7 categories and 26 Claw-style operations, such as web retrieval, file processing, and text editing.

Seed-Guided Task Instruction Generation. Given z, a task generator $\mathcal{M}_{\text{task}}$ generates the user-facing instruction via the prompt template $\pi(\cdot)$:

$$p = \mathcal{M}_{\text{task}}(\pi(z)). \tag{1}$$

The persona and scenario define the user intent, while the selected operations constrain the task to feasible workspace actions, yielding diverse multi-step instructions grounded in realistic user needs.

3.1.2 Skill-Grounded Bottom-Up Synthesis

Skill-grounded bottom-up synthesis constructs tasks from reusable skills collected from ClawHub (OpenClaw, 2026a). Instead of treating each skill as a complete task, we regard it as a capability unit that can be annotated, filtered, and composed into environment-grounded workflows.

Skill Annotation and Filtering. Given a raw skill collection $\mathcal{K} = \{k_i\}_{i=1}^{N}$, $\mathcal{M}_{\text{ann}}$ converts each skill into a structured annotation:

$$\alpha_i = \mathcal{M}_{\text{ann}}(\rho(k_i)), \tag{2}$$

where $\rho(\cdot)$ is the annotation prompt, and α_i summarizes the skill's functionality, constraints, input-output characteristics, and synthesizability. Skills that require unavailable credentials, lack sufficient task-relevant details, or are incompatible with our task format are filtered out. This process retains 16K synthesizable skills from an initial pool of approximately 30K, providing normalized capability descriptions for downstream task generation.

Skill-Composition Task Construction. After filtering, each task is constructed from one primary skill and up to three supporting skills. The primary skill defines the main objective, while supporting skills introduce auxiliary operations or contextual requirements. The task generator $\mathcal{M}_{\text{task}}$ conditions on the selected skills and their annotations α to produce a user-facing instruction:

$$p = \mathcal{M}_{\text{task}}\big(\pi(k_{\text{main}}, \mathcal{K}_{\text{support}}, \alpha)\big), \tag{3}$$

where k_{main} is the primary skill and $\mathcal{K}_{\text{support}}$ is the supporting skill set. This design turns skills into composable capabilities for task construction.

3.2 Resource Preparation

Claw-style tasks require concrete workspace resources as part of the initial state. Given a synthesized instruction p, resource preparation identifies the required paths, formats, schemas, fields, and content constraints, summarized as $f = \{(l_i, t_i, d_i)\}_{i=1}^{m}$, where l_i, t_i, and d_i denote the file path, type, and content specification. A resource generator then materializes f into task-specific files, such as documents, tables, configuration files, or reference materials, and places them at the specified workspace paths. The generated resources are designed to be self-contained and verifier-compatible. Text files provide the entities and constraints required by the instruction, while structured files contain explicit values that allow

checkers to recompute statistics or validate output consistency. This controlled construction makes each task executable and reproducible while avoiding privacy risks from real user files.

3.3 Verification Design

Verification checks whether the final workspace state satisfies the user instruction. Since Claw-style tasks involve both objective workspace changes and qualitative requirements, we adopt hybrid code-based and rubric-based verification (PinchBench, 2026). A verifier generator produces executable checkers and rubric rules for each task instance.

Code-Based Verification. Code-based checkers evaluate deterministic requirements, such as file existence, schema validity, numerical correctness, and input-output consistency. They also capture implicit constraints from task resources, such as recomputing statistics or validating eligible records. These checks produce the code-based score s_{code}.

Rubric-Based Verification. Rubric-based verification complements code checks by assessing qualitative requirements such as clarity, completeness, and faithfulness. Each rubric assigns a discrete score from $\{0, 0.25, 0.5, 0.75, 1.0\}$, and the rubric-based score s_{rubric} is computed as the weighted average over all rubric rules.

Score Aggregation. For tasks with only code-based verification, we set $s_{\text{task}} = s_{\text{code}}$. For hybrid tasks, we combine the two signals as:

$$s_{\text{task}} = \lambda s_{\text{code}} + (1 - \lambda) s_{\text{rubric}}, \tag{4}$$

where $\lambda = 0.7$. This weighting prioritizes concrete workspace changes while still capturing qualitative requirements that cannot be checked by code alone.

3.4 Automated Quality Assessment

After task synthesis and verification design, we filter low-quality samples by assessing task and verifier quality.

3.4.1 Task Quality Assessment

We assess task quality along three dimensions: novelty, plausibility, and difficulty.

Novelty. We measure novelty by computing the maximum cosine similarity between a new task and the retained task pool. Tasks whose similarity exceeds a threshold 0.8 are filtered out to reduce redundancy and improve scenario coverage.

Plausibility. Task plausibility is assessed with an LLM judge by checking whether each task is clear, internally consistent, and realistic as a Claw-style user request. Tasks relying on unavailable tools, unsupported services, or unrealistic environment assumptions are removed.

Difficulty. Task difficulty is estimated based on the expected number of steps, operation diversity, and required planning or cross-resource reasoning. These scores help maintain a balanced mixture of simple, moderate, and challenging tasks.

3.4.2 Verification Quality Assessment

We assess both code-based and rubric-based verifiers to reduce verifier-induced noise, including non-executable checkers, overly permissive scoring, and over-strict scoring. Code checkers must be executable and requirement-aligned, while rubric rules should provide complementary qualitative signals without duplicating deterministic checks.

Code Assessment. For code checkers, we first run an executable sanity check to ensure that the checker is syntactically valid and can run without errors. To detect overly permissive checkers, we execute the checker on the initial workspace without any agent-produced outputs; checkers that assign non-trivial scores in this state are filtered out. We then use an LLM judge to review requirement coverage and over-strictness, ensuring that the checker covers core objective requirements without enforcing constraints beyond the task specification.

Rubric Assessment. For rubric rules, we assess whether they complement the code checker. Since deterministic checks already cover objective requirements, rubrics should focus on qualitative aspects such as tone, clarity, organization, faithfulness, and completeness. Rubrics that merely duplicate code-verifiable requirements are removed to avoid overweighting simple objective constraints.

3.5 Synthesized Task Analysis

Figure 4 and Table 6 show that ClawGym-SynData covers diverse scenarios, operations, and reusable capabilities. In persona-driven synthesis, no scenario category dominates the distribution, with the largest category accounting for only 12.5% of tasks. In skill-grounded synthesis, the 16K retained skills span broad capability categories, including Data & APIs, Dev Tools, Workflows, Automation, Security, Prompts, and MCP Tools. These results suggest

that the two synthesis routes provide complementary diversity: personas expand user-scenario coverage, while reusable skills strengthen operational grounding. We further provide a human-sampled quality assessment in Appendix A.1, showing that the synthesized tasks are generally coherent, executable, and verifier-compatible.

4 ClawGym-Agents: Training Claw Agents with Synthesized Tasks

This section presents the ClawGym-Agents training pipeline, which collects black-box rollouts, aggregates fragmented interaction logs into multi-turn trajectories, and filters them for agent training.

4.1 Black-Box Rollout

The training paradigm for Claw agents differs from traditional search or coding agents (Sun et al., 2026; Song et al., 2026a). Unlike conventional agentic training with explicit and predefined agent loops, OpenClaw is highly encapsulated, with internal execution details such as context management and subagent sessions hidden from direct access.

To collect authentic trajectories at scale, we execute synthesized tasks directly through the OpenClaw harness rather than reimplementing its agent loop. We deploy multiple Docker instances on a distributed cluster, assign each task and its workspace materials to an isolated instance, and invoke the OpenClaw runtime for task execution. During rollout, a proxy layer records the multi-turn request-response stream without modifying the agent logic, capturing model inputs, outputs, tool calls, and environment feedback. We use MiniMax-M2.5 (MiniMax, 2026a) and GLM-5.1 (Z.ai, 2026) as teacher models, and the resulting trajectories serve as the basis for aggregation, filtering, and training.

4.2 Trajectory Aggregation and Selection

After rollout collection, we aggregate raw request logs into complete trajectories. Since the proxy records individual model calls, one task execution may produce multiple partially overlapping traces. We reconstruct coherent multi-turn sequences by grouping requests with identical message prefixes and concatenating subsequent interaction turns. We also remove systematic auxiliary prompts, such as cron or heartbeat messages, that maintain session activity but do not contribute to task completion.

We then select trajectories using deterministic code-based verifier scores, which provide scalable and lower-noise filtering signals. Since these scores are continuous in [0, 1], binary best-of-N filtering strategies (Song et al., 2026a) are not directly suitable. Instead, we retain valid and complete trajectories whose scores exceed a predefined reward threshold. This threshold-based strategy balances completion quality and data coverage, yielding 24.5K high-fidelity interaction trajectories. Section 6.3 analyzes the effect of different thresholds.

	Rounds	Tokens	Tool Calls	Tool Types
Overall	13.00	18.67K	15.82	3.25

Table 1: Statistics of selected trajectories after filtering.

As shown in Table 1, the filtered trajectories remain substantially multi-turn and tool-intensive, averaging 13.00 interaction rounds, 18.67K tokens, 15.82 tool calls, and 3.25 distinct tool types per trajectory. Using these trajectories, we train Qwen3-family models to obtain **ClawGym-Agents**. Training details are provided in Appendix B.

5 ClawGym-Bench: A Reliable Benchmark for Claw Agents

To support reliable evaluation, we construct **ClawGym-Bench** from the synthesized task pool through a more fine-grained selection process, which combines automated filtering with human review to remove noisy samples and retain reliable workspace-grounded tasks across scenarios.

5.1 Benchmark Construction

We construct ClawGym-Bench from the synthesized task pool after task-quality and verifier-quality filtering. Since benchmark samples directly support model comparison, we apply a stricter construction process focusing on difficulty calibration, verifier reliability, and scenario coverage.

Difficulty-Aware Filtering. Let $\mathcal{D}_{\text{cand}}$ denote the candidate task pool after quality filtering in Section 3.4. To select discriminative benchmark samples, we perform rollout-based difficulty filtering. For each task $\tau \in \mathcal{D}_{\text{cand}}$, we run $n = 4$ rollouts with both a strong and a smaller LLM, and compute their average completion scores $\bar{s}_{\text{strong}}(\tau)$ and $\bar{s}_{\text{small}}(\tau)$, retaining a task only if:

$$\begin{cases} \bar{s}_{\text{strong}}(\tau) \geq 0.2, \\ \bar{s}_{\text{small}}(\tau) \leq 0.6, \\ \bar{s}_{\text{strong}}(\tau) > \bar{s}_{\text{small}}(\tau). \end{cases} \tag{5}$$

These criteria prioritize verifiable solvability and discriminative power, while the final human-LLM review further helps maintain task validity.

LLM-Assisted Human Review. After automated filtering and difficulty-aware selection, the remaining candidates undergo a final human-LLM review. A frontier LLM judge first diagnoses potential issues in the instruction, resources, checker, and optional rubric rules, focusing on task clarity, resource consistency, checker executability and strictness, and rubric complementarity. Human reviewers then verify the feedback, decide whether to accept, revise, or reject each candidate, and add missing details when necessary. This process improves review efficiency while keeping final benchmark decisions under human oversight.

5.2 Benchmark Composition

The final benchmark contains 200 tasks, each with a user instruction, mock resources, and a verifier. Among them, 156 tasks use code-only verification and 44 use hybrid verification with rubric-based judgment. The benchmark spans six task categories (Table 11), and all tasks are de-duplicated against the training set to avoid train-test overlap.

ClawGym-Bench evaluates agents in a harness-based workspace setting, where each task is initialized with mock resources and scored by its verifier after execution. Since such evaluation involves multi-turn interactions, tool calls, and workspace updates, repeated runs are more costly than text-only scoring. We therefore assess reliability from two aspects. For *evaluation stability*, models are evaluated 10 times on the benchmark; the small standard deviations (< 1%) in Table 2 indicate stable comparisons under repeated runs. For *verifiable solvability*, each task is checked through successful agent rollouts or human-constructed reference completions to ensure a feasible path to full credit, reducing the risk that failures come from infeasible task design or unreachable verification criteria.

Model	Runs	Mean Score(%)	Std.(%)
Qwen3-8B	10	36.1	0.28
Qwen3-30B-A3B	10	44.9	0.98

Table 2: Stability analysis under repeated evaluation.

6 Experiment

In this section, we evaluate our ClawGym framework by extensively examining the data quality, training effectiveness, and benchmark reliability.

6.1 Experimental Settings

Evaluation. We evaluate models on ClawGym-Bench and three out-of-domain benchmarks: PinchBench (PinchBench, 2026), Claw-Eval (Ye et al., 2026), and τ^2-Bench (Barres et al., 2025). For ClawGym-Bench, code-only tasks use the code-based score directly, while hybrid tasks combine code-based and rubric-based scores with weights of 0.7 and 0.3, respectively. For PinchBench, we use the April 10, 2026 release and exclude multimodal tasks; for Claw-Eval and τ^2-Bench, we follow the official evaluation protocols. When rubric judgment is needed, we use GPT-5.4 with prompts listed in Appendix E.4. For ClawGym-Bench and PinchBench, trajectories are collected through black-box rollouts in the original OpenClaw environments, with a uniform context length of 64K tokens.

Baselines. We evaluate models across different capability levels to assess the discriminative ability of ClawGym-Bench and the gains from ClawGym-SynData training. The models include proprietary systems such as Claude-4.7-Opus (Anthropic, 2026b), Claude-4.6-Opus (Anthropic, 2026a), Claude-4.6-Sonnet (Anthropic, 2026c), Gemini-3-Flash (Google Cloud, 2026), and GPT-5.4 (OpenAI, 2026), as well as open-weight models including DeepSeek-V3.2 (Liu et al., 2025), GLM-5.1 (Z.ai, 2026), MiniMax-M2.7 (MiniMax, 2026b), Kimi-K2.6 (Moonshot AI, 2026), Qwen3.5-Plus (QwenTeam, 2026), and Qwen3-series models (Yang et al., 2025). We also include general agentic baselines, including EnvScaler-8B (Song et al., 2026b) and AWM-8B/14B (Wang et al., 2026b).

6.2 Main Results

We evaluate ClawGym from three perspectives: training effectiveness, benchmark discriminativeness, and cross-benchmark transfer.

• *Effectiveness of Synthesized Data.* Training on trajectories derived from ClawGym-SynData substantially improves compact open-weight backbones on ClawGym-Bench. As shown in Table 3, ClawGym-8B improves over Qwen3-8B from 35.02 to 50.24, and ClawGym-30A3B improves over Qwen3-30A3B from 45.11 to 56.82. Notably, ClawGym-30A3B also surpasses the much larger Qwen3-235A23B, suggesting that high-quality agent-specific trajectories can partially

Model	Product. & Collab.	Systems & Auto.	Analysis & Reason.	Content & Domain	Planning & Knowl.	Software Dev.	Avg.
Proprietary Frontier Models							
Claude-4.7-Opus	73.02	**74.28**	**80.14**	**80.98**	**81.04**	82.04	**77.81**
Claude-4.6-Sonnet	70.44	68.04	76.27	74.59	74.21	73.40	72.40
Claude-4.6-Opus	68.02	63.49	73.23	74.40	73.38	82.90	71.43
Gemini-3-Flash	68.70	68.06	68.89	78.87	80.68	**84.04**	73.50
GPT-5.4	**76.00**	66.56	73.83	76.19	68.51	82.40	73.49
Open-Weight Frontier Models							
DeepSeek-V3.2	64.45	64.43	71.97	72.30	66.66	70.34	67.90
GLM-5.1	**70.95**	64.11	71.71	**75.99**	**74.09**	73.86	**71.12**
MiniMax-M2.7	63.98	60.11	63.96	63.42	68.72	64.10	63.72
Kimi-K2.6	67.16	55.99	72.56	72.84	71.56	**82.47**	69.05
Qwen3.5-Plus	67.34	**66.56**	**74.27**	73.07	71.79	72.10	70.35
Compact Open-Weight Models							
Qwen3-8B	37.46	29.06	30.40	41.12	44.47	30.69	35.02
Qwen3-32B	40.68	36.21	37.84	47.16	49.29	33.11	40.32
Qwen3-30A3B	42.47	42.09	45.04	51.95	45.98	46.24	45.11
Qwen3-235A23B	**53.66**	**52.27**	47.18	61.39	**67.33**	49.23	54.48
ClawGym-4B	45.21	46.60	47.51	48.99	52.25	48.27	47.73
ClawGym-8B	49.47	46.83	46.35	55.37	52.29	54.78	50.24
ClawGym-30A3B	52.98	50.97	**64.64**	**61.46**	57.90	**56.13**	**56.82**

Table 3: Performance comparison of different models on ClawGym-Bench. The best and second results within each model group are highlighted in **bold** and underlined, respectively.

Model	Pinch.	Claw-Eval	τ^2-Bench	Avg.
Qwen3-8B	54.50	23.12	36.11	37.91
EnvScaler-8B	48.23	25.26	32.89	35.46
AWM-8B	47.27	23.62	33.80	34.90
AWM-14B	54.77	27.14	**38.53**	40.15
ClawGym-8B	**75.70**	**29.03**	36.77	**47.17**

Table 4: Comparison with general agentic baselines on external benchmarks. Pinch. denotes PinchBench.

compensate for model scale in Claw-style environments. These results demonstrate that ClawGym-derived trajectories serve as effective supervision for improving Claw-style agent capabilities.

• *Discriminative Capacity of ClawGym-Bench.* ClawGym-Bench provides clear discriminative signals across both capability levels and task categories. The average scores span a wide range, from 35.02 for Qwen3-8B to 77.81 for Claude-4.7-Opus, showing that the benchmark avoids early saturation while distinguishing models at different capability levels. The category-wise results further reveal diverse model strengths: Claude-4.7-Opus achieves the highest overall score, while GPT-5.4 performs best on *Product. & Collab.* and Gemini-3-Flash leads on *Software Dev.* These results indicate that ClawGym-Bench supports both aggregate comparison and fine-grained diagnosis of agent capabilities.

• *Cross-Benchmark Transfer.* To examine cross-benchmark transfer, we compare ClawGym-8B with its backbone and general agentic baselines on external benchmarks. As shown in Table 4, ClawGym-8B achieves the best average score, with strong gains on PinchBench and consistent improvements over same-size baselines across all three benchmarks. These results suggest that ClawGym-derived trajectories provide useful supervision beyond ClawGym-Bench.

6.3 Further Analysis

In this section, we provide additional analysis to better understand the training behavior of ClawGym-Agents. Specifically, we evaluate the interaction between the two synthesis routes, analyze training dynamics, and examine the impact of reward-based trajectory filtering.

Synergy of Synthesis Strategies. To assess the necessity of persona-driven top-down synthesis and skill-grounded bottom-up synthesis, we compare models trained on trajectories collected from different task sources. *Persona* and *Skill* use trajectories from the corresponding strategy alone, while *Mixed* combines both sources. As shown in Table 5, models trained with mixed synthesis outperform those trained with either strategy alone, highlighting the synergy between top-down intent diversity and bottom-up operational grounding. Specifically, the persona-driven strategy ensures a broad cov-

erage of realistic user scenarios, while the skill-grounded strategy anchors tasks in the agent's actual functional boundaries. By combining these, the model learns to bridge the gap between abstract user needs and concrete tool execution, leading to better generalization across diverse tasks.

Base Model	Source	ClawGym-Bench	Pinch.
Qwen3-8B	*Persona*	49.44	73.51
	Skill	49.06	68.23
	Mixed	**50.24**	**75.68**
Qwen3-30A3B	*Persona*	53.65	84.92
	Skill	52.27	80.05
	Mixed	**56.82**	**86.00**

Table 5: Synergy between persona-driven and skill-grounded synthesis strategies.

Training Dynamics and Convergence. To examine how training exposure affects model performance, we analyze checkpoints across 5 epochs (103 steps per epoch), evaluating every 60 steps. As shown in Figure 2, performance on both ClawGym-Bench and PinchBench improves rapidly in the early stages and peaks near the end of the third epoch (step 309). Afterward, performance slightly declines, suggesting that excessive exposure to the synthesized trajectory distribution may lead to mild overfitting or reduced generalization. These results highlight the importance of selecting an appropriate scale for Claw agent training.

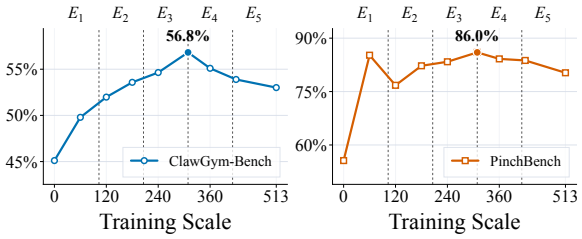


Figure 2: Effect of training scale on Qwen3-30A3B using ClawGym-SynData. E_i denotes the i-th epoch.

Impact of Reward-Based Trajectory Filtering. As discussed in Section 4.2, we select training trajectories using code-based verifier scores, which provide deterministic filtering signals for trajectory quality. We evaluate reward thresholds from 0.4 to 0.9, and Figure 3 shows that 0.5 performs best on both benchmarks. Lower thresholds introduce incomplete trajectories, while higher thresholds reduce behavioral diversity, including partially successful recovery or intermediate strategies. These results show that calibrated code-based filtering is important for balancing completion quality and data diversity in trajectory selection.

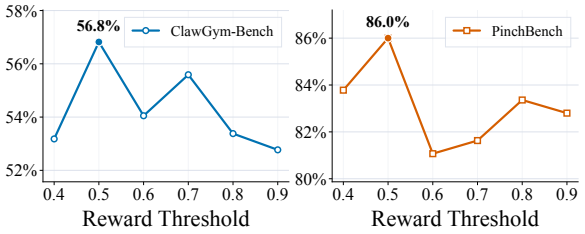


Figure 3: Effect of trajectory filtering reward threshold on Qwen3-30A3B trained with ClawGym-SynData.

7 Behavioral Analysis of Claw Agents

Claw-style tasks differ from prior text-only or structured agentic tasks because agents must act over local states, tools, execution feedback, and persistent workspace artifacts. This setting introduces three key capability challenges: using tools as coherent workflows rather than isolated calls; recovering from inevitable errors in long-horizon interactions before they compound into task failure; and preserving fine-grained constraints such as schemas, filtering rules, and cross-file consistency in generated artifacts. We analyze representative agent behaviors along these dimensions, with detailed case studies provided in Appendix D. These observations may offer useful insights for future work on data synthesis, agent training, and verifier design.

8 Conclusion

In this paper, we propose ClawGym, a scalable framework for developing Claw-style personal agents by unifying task synthesis, agentic training, and diagnostic evaluation. ClawGym constructs ClawGym-SynData, a large-scale dataset of executable and verifiable tasks generated through persona-driven and skill-grounded synthesis with realistic mock workspaces and hybrid verification. Based on these tasks, we collect black-box rollout trajectories to train ClawGym-Agents, and further build ClawGym-Bench for reliable evaluation across diverse workspace-grounded scenarios. Experiments show that ClawGym-SynData yields consistent gains across model scales, while ClawGym-Bench clearly distinguishes agents with different capabilities. These findings position ClawGym as a step toward scalable, verifiable, and reproducible development of personal agents.

Limitations

Despite our efforts to build a scalable framework for Claw-style personal agents, this work still has several limitations. First, due to computational constraints, our training experiments focus on relatively small and medium-scale Qwen3-family models, leaving the exploration of substantially larger backbones to future work. Second, although ClawGym-Bench is designed to cover diverse workspace-grounded scenarios, its difficulty-aware filtering prioritizes verifiable solvability and discriminative power, so the overall task difficulty remains moderate. More challenging long-horizon tasks with stronger dependencies across tools, files, and sessions should be further studied. Third, for tasks evaluated with continuous non-binary rewards, our current trajectory selection relies on a simple threshold-based filtering strategy. Future work may explore more principled data selection methods that better exploit fine-grained reward signals for agent training.

Ethics Statement

We strictly follow research ethics and use only publicly available resources, synthetic tasks, and mock workspaces under their intended research usage. ClawGym is designed to support benign personal-agent development, such as productivity, information processing, file management, and daily digital workflows, rather than harmful automation, privacy invasion, or malicious tool use. We do not intentionally collect private, sensitive, or personally identifiable information; the constructed workspaces are synthetic and are used only for controlled research evaluation and training. AI assistance was used only during the paper refinement process.

References

AgentScope-AI. 2026. QwenPaw: Your personal ai assistant. https://github.com/agentscope-ai/QwenPaw. GitHub repository. Accessed: 2026-04-29.

Anthropic. 2026a. Introducing claude opus 4.6. https://www.anthropic.com/news/claude-opus-4-6. 2026-02-05.

Anthropic. 2026b. Introducing claude opus 4.7. https://www.anthropic.com/news/claude-opus-4-7. 2026-04-16.

Anthropic. 2026c. Introducing claude sonnet 4.6. https://www.anthropic.com/news/claude-sonnet-4-6. 2026-02-17.

Fei Bai, Yingqian Min, Beichen Zhang, Zhipeng Chen, Wayne Xin Zhao, Lei Fang, Zheng Liu, Zhongyuan Wang, and Ji-Rong Wen. 2025. Towards effective code-integrated reasoning. *Preprint*, arXiv:2505.24480.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. 2025. τ^2-bench: Evaluating conversational agents in a dual-control environment. *Preprint*, arXiv:2506.07982.

Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, and 1 others. 2024. Windows agent arena: Evaluating multi-modal os agents at scale. *arXiv preprint arXiv:2409.08264*.

Kaiyuan Chen, Qimin Wu, Taiyu Hou, Tianhao Tang, Xueyu Hu, Yuchen Hou, Bikun Li, Chengming Qian, Guoyin Wang, Haolin Chen, and 1 others. 2026. Agentif-oneday: A task-level instruction-following benchmark for general ai agents in daily scenarios. *arXiv preprint arXiv:2601.20613*.

Daixuan Cheng, Shaohan Huang, Yuxian Gu, Huatong Song, Guoxin Chen, Li Dong, Wayne Xin Zhao, Ji-Rong Wen, and Furu Wei. 2026. Llm-in-sandbox elicits general agentic intelligence. *arXiv preprint arXiv:2601.16206*.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. 2024. Introducing swe-bench verified.

Shuangrui Ding, Xuanlang Dai, Long Xing, Shengyuan Ding, Ziyu Liu, Jingyi Yang, Penghui Yang, Zhixiong Zhang, Xilin Wei, Yubo Ma, Haodong Duan, Jing Shao, Jiaqi Wang, Dahua Lin, Kai Chen, and Yuhang Zang. 2026. Wildclawbench.

Guanting Dong, Junting Lu, Junjie Huang, Wanjun Zhong, Longxiang Liu, Shijue Huang, Zhenyu Li, Yang Zhao, Xiaoshuai Song, Xiaoxi Li, and 1 others. 2026. Agent-world: Scaling real-world environment synthesis for evolving general agent intelligence. *arXiv preprint arXiv:2604.18292*.

Guanting Dong, Hangyu Mao, Kai Ma, Licheng Bao, Yifei Chen, Zhongyuan Wang, Zhongxia Chen, Jiazhen Du, Huiyang Wang, Fuzheng Zhang, Guorui Zhou, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. 2025. Agentic reinforced policy optimization. *CoRR*, abs/2507.19849.

Google Cloud. 2026. Gemini 3 Flash: Generative ai on vertex ai. https://docs.cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/3-flash. Model documentation. Accessed: 2026-04-29.

HKUDS. 2026. NanoBot: The ultra-lightweight personal ai agent. https://github.com/HKUDS/

nanobot. GitHub repository. Accessed: 2026-04-29.

Junlong Li, Wenshuo Zhao, Jian Zhao, Weihao Zeng, Haoze Wu, Xiaochen Wang, Rui Ge, Yuxuan Cao, Yuzhen Huang, Wei Liu, and 1 others. 2025. The tool decathlon: Benchmarking language agents for diverse, realistic, and long-horizon task execution. *arXiv preprint arXiv:2510.25726*.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, and 1 others. 2025. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Ziyu Ma, Shidong Yang, Yuxiang Ji, Xucong Wang, Yong Wang, Yiming Hu, Tongwen Huang, and Xiangxiang Chu. 2026. SkillClaw: Let skills evolve collectively with agentic evolver. *arXiv preprint arXiv:2604.08377*.

MAA. 2026. American Invitational Mathematics Examination - AIME 2026.

Fanqing Meng, Lingxiao Du, Zijian Wu, Guanzheng Chen, Xiangyan Liu, Jiaqi Liao, Chonghe Jiang, Zhenglin Wan, Jiawei Gu, Pengfei Zhou, Rui Huang, Ziqi Zhao, Shengyuan Ding, Ailing Yu, Bo Peng, Bowei Xia, Hao Sun, Haotian Liang, Ji Xie, and 28 others. 2026. Clawmark: A living-world benchmark for multi-turn, multi-day, multimodal coworker agents. *Preprint*, arXiv:2604.23781.

MiniMax. 2026a. Minimax m2.5: Built for real-world productivity. https://www.minimax.io/news/minimax-m25. 2026-02-16.

MiniMax. 2026b. MiniMax M2.7: Model self-improvement, driving productivity innovation through technological breakthroughs. https://www.minimax.io/models/text/m27. Official model page. Accessed: 2026-04-29.

Moonshot AI. 2026. Kimi k2.6: From code to creation, from one to many. https://www.kimi.com/blog/kimi-k2-6. 2026-04-07.

OpenAI. 2026. Introducing GPT-5.4. https://openai.com/index/introducing-gpt-5-4/. Official release announcement. Accessed: 2026-04-29.

OpenClaw. 2026a. ClawHub: Skill registry for openclaw. https://clawhub.ai/. Online skill registry. Accessed: 2026-04-29.

OpenClaw. 2026b. OpenClaw: Personal ai assistant. https://github.com/openclaw/openclaw. GitHub repository. Accessed: 2026-04-29.

Tejal Patwardhan, Rachel Dias, Elizabeth Proehl, Grace Kim, Michele Wang, Olivia Watkins, Simón Posada Fishman, Marwan Aljubeh, Phoebe Thacker, Laurance Fauconnet, Natalie S. Kim, Patrick Chao, Samuel Miserendino, Gildas Chabot, David Li, Michael Sharman, Alexandra Barr, Amelia Glaese, and Jerry Tworek. 2025. Gdpval: Evaluating ai model performance on real-world economically valuable tasks. *Preprint*, arXiv:2510.04374.

Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. 2023. Yarn: Efficient context window extension of large language models. *Preprint*, arXiv:2309.00071.

PinchBench. 2026. PinchBench: Real-world benchmarks for ai coding agents. https://github.com/pinchbench/skill. GitHub repository. Accessed: 2026-04-29.

QwenTeam. 2026. Qwen3.5: Towards native multimodal agents. https://qwen.ai/blog?id=qwen3.5. 2026-02-16.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Huatong Song, Lisheng Huang, Shuang Sun, Jinhao Jiang, Ran Le, Daixuan Cheng, Guoxin Chen, Yiwen Hu, Zongchao Chen, Yiming Jia, and 1 others. 2026a. Swe-master: Unleashing the potential of software engineering agents via post-training. *arXiv preprint arXiv:2602.03411*.

Xiaoshuai Song, Haofei Chang, Guanting Dong, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. 2026b. Envscaler: Scaling tool-interactive environments for llm agent via programmatic synthesis. *arXiv preprint arXiv:2601.05808*.

Shuang Sun, Huatong Song, Lisheng Huang, Jinhao Jiang, Ran Le, Zhihao Lv, Zongchao Chen, Yiwen Hu, Wenyang Luo, Wayne Xin Zhao, and 1 others. 2026. Swe-world: Building software engineering agents in docker-free environments. *arXiv preprint arXiv:2602.03419*.

Shuang Sun, Huatong Song, Yuhao Wang, Ruiyang Ren, Jinhao Jiang, Junjie Zhang, Fei Bai, Jia Deng, Wayne Xin Zhao, Zheng Liu, Lei Fang, Zhongyuan Wang, and Ji-Rong Wen. 2025a. Simpledeepsearcher: Deep information seeking via web-powered reasoning trajectory synthesis. *Preprint*, arXiv:2505.16834.

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

Xinyu Tang, Han Peng, Guoxin Chen, Yuze Shi, Zitao Su, Peiyu Liu, Wayne Xin Zhao, Yawen Li, and Zhe Xue. 2026. Agent systems with harness engineering: A systematic survey.

Daoyu Wang, Qingchuan Li, Jie Ouyang, Shuo Yu, Mingyue Cheng, and Qi Liu. 2025. Claw-r1: The data foundation for agentic reinforcement learning. https://github.com/AgentR1/Claw-R1. GitHub repository.

Yinjie Wang, Xuyang Chen, Xiaolong Jin, Mengdi Wang, and Ling Yang. 2026a. Openclaw-rl: Train any agent simply by talking. *arXiv preprint arXiv:2603.10165*.

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

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

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, and 1 others. 2026a. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning. *arXiv preprint arXiv:2602.08234*.

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. 2026b. Metaclaw: Just talk an agent that meta-learns and evolves in the wild. *arXiv preprint arXiv:2603.17187*.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, and 1 others. 2024. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Qianyu Yang, Yang Liu, Jiaqi Li, Jun Bai, Hao Chen, Kaiyuan Chen, Tiliang Duan, Jiayun Dong, Xiaobo Hu, Zixia Jia, and 1 others. 2026. Onemillion-bench: How far are language agents from human experts? *arXiv preprint arXiv:2603.07980*.

Bowen Ye, Rang Li, Qibin Yang, Yuanxin Liu, Linli Yao, Hanglong Lv, Zhihui Xie, Chenxin An, Lei Li, Lingpeng Kong, and 1 others. 2026. Claw-eval: Toward trustworthy evaluation of autonomous agents. *arXiv preprint arXiv:2604.06132*.

Z.ai. 2026. Glm-5.1: Towards long-horizon tasks. https://z.ai/blog/glm-5.1. 2026-04-07.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, and 1 others. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2).

Jiachen Zhu, Menghui Zhu, Renting Rui, Rong Shan, Congmin Zheng, Bo Chen, Yunjia Xi, Jianghao Lin, Weiwen Liu, Ruiming Tang, Yong Yu, and Weinan Zhang. 2025. Evolutionary perspectives on the evaluation of llm-based ai agents: A comprehensive survey. *arXiv preprint arXiv:2506.11102*.

A Dataset Analysis

A.1 Task Synthesis

Seed Preparation. The category and atomic action set used for task synthesis are shown in Table 9 and Table 10.

Task Distribution. The scenario category distribution and atomic action distribution of synthesized tasks are shown in Figure 4; and the category distribution of the annotated skills collected for synthesis is shown in Table 6. During the synthesis, GPT-5 is used for task generation, skill annotation, resource preparation, and verifier generation.

Human-Sampled Quality Assessment. We conduct a stratified human audit on 200 samples from ClawGym-SynData as a complementary assessment of task-generation quality. The samples are drawn proportionally to the overall data distribution while ensuring coverage of synthesis sources, scenario categories, task difficulties, and verifier types. Each task is rated on a 1–5 scale along four dimensions: task reasonableness, execution feasibility, resource consistency, and verification quality. These dimensions respectively assess whether a task is realistic and clear, executable under the provided workspace, consistent with its mock resources, and paired with an appropriate verifier. As shown in Table 7, the sampled tasks achieve an overall score of 4.34/5, with consistently strong ratings across task-side and verifier-side dimensions. These results provide additional evidence that the synthesis pipeline can produce coherent, executable, and verifier-compatible training data.

A.2 Benchmark Data

The category distribution of ClawGym-Bench is shown in Table 11. The six evaluation categories group the fine-grained synthesis taxonomy into broader categories for stable category-wise reporting. For ClawGym-Bench construction, GPT-5.4 is used as the LLM judge in the human-LLM review stage to diagnose potential issues in task instructions, resources, checkers, and rubric rules. All benchmark inclusion decisions are ultimately verified by human reviewers.

A.3 Evaluation Details

For ClawGym-Bench, we report the single-run average verifier score, scaled to percentage points, because the benchmark shows stable evaluation results under repeated runs. For PinchBench, we report Avg@3. For Claw-Eval, we report Pass@3, following the official setting. For τ^2-Bench, we evaluate the retail, telecom, and airline domains and report Avg@3 by averaging the scores over the three domains.

B Training Details for Claw Agents

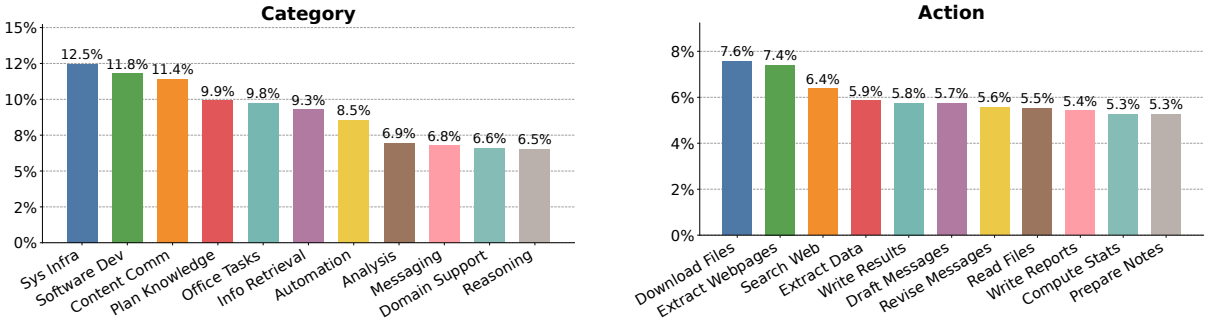
Using the filtered trajectories, we perform multi-turn SFT on the Qwen3-series (Yang et al., 2025), including Qwen3-4B-2507-Instruct, Qwen3-8B, and Qwen3-30B-A3B-2507-Instruct. Since Claw tasks often require long-horizon interaction with local files, tools, and environment feedback, the resulting trajectories can substantially exceed the context length used in standard instruction tuning. For Qwen3-8B, whose native context window is 32K tokens, we apply YaRN (Peng et al., 2023) to extend the maximum context length to 64K tokens, enabling the model to better capture long-range dependencies across multi-turn agent trajectories.

During training, we adopt a multi-turn loss masking strategy tailored to agentic interaction data. Specifically, tokens corresponding to environment feedback produced by Docker-based execution are excluded from the supervised loss. This prevents the model from learning to imitate deterministic execution outputs or tool-returned observations, which are not actions generated by the policy. Instead, the optimization objective focuses on the model-generated parts of the trajectory, including reasoning, decision making, and tool-call generation. This design encourages the model to learn how to act within the environment rather than memorize the environment's responses.

Additional comparisons with general agentic baselines on ClawGym-Bench are provided in Table 8. These results further show the effectiveness of ClawGym-derived training for improving Claw-style agent capabilities.

C Preliminary Sandbox-Parallel RL Exploration

This appendix provides a preliminary feasibility study of applying outcome-reward reinforcement learning in ClawGym. Since RL is not the main focus of this paper, we include these results only to illustrate the extensibility of the ClawGym framework rather than as primary evidence for the main claims. The main experiments focus on synthesized task construction, black-box trajectory collection, supervised fine-tuning, and benchmark evaluation.



(a) Scenario category distribution.

(b) Atomic action distribution.

Figure 4: Task distribution of persona-driven synthesis. Figure (a) shows the distribution of user-facing scenario categories, while Figure (b) shows the distribution of atomic actions required by the generated tasks. Together, they characterize both the topical diversity and executable-operation diversity of the synthesized tasks.

Category	MCP Tools	Prompts	Workflows	Dev Tools	Data & APIs	Security	Automation	Other	Total
Count	411	565	1972	3906	4236	993	1221	3533	16837
Percentage	2.44%	3.36%	11.71%	23.20%	25.16%	5.90%	7.25%	20.98%	100.00%

Table 6: Category distribution of the annotated skills collected for synthesis.

Metric	Avg Score
Task Reasonableness	4.42
Execution Feasibility	4.33
Resource Consistency	4.36
Verification Quality	4.25
Overall	4.34

Table 7: Human quality audit on 200 stratified samples from ClawGym-SynData. Samples are drawn proportionally to the overall data distribution, and each metric is scored on a 1-5 scale.

Model	**ClawGym-Bench**
Qwen3-8B	35.02
EnvScaler-8B	39.56
AWM-8B	34.52
AWM-14B	42.97
ClawGym-8B	**50.24**

Table 8: Comparison with general agentic baselines on ClawGym-Bench.

Beyond supervised fine-tuning, we further explore reinforcement learning for Claw-style agents through a lightweight sandbox-parallel pipeline. The pipeline keeps the OpenClaw agent loop as a black box and virtualizes each task into an independent sandbox containing its own root filesystem, workspace, gateway, and verifier (Cheng et al., 2026). Our design has three practical advantages: (i) *sandbox-parallel rollout*, which allows many task rollouts to run concurrently without interfering with each other; (ii) *low infrastructure dependency*, enabled by interchangeable Docker-based and Docker-free sandbox backends for different cluster settings; and (iii) *outcome-reward-only training*, where the code verifier provides the reward signal directly, avoiding auxiliary reward models or process-level supervision.

We conduct RL from two starting points with different capabilities: Qwen3-4B-2507-Instruct, a vanilla instruct model without ClawGym fine-tuning, and ClawGym-30B-A3B, a SFT model initialized from Qwen3-30B-A3B-2507-Instruct, using ClawGym-SynData. For training, we sample 2,000 tasks from ClawGym-SynData and balance the selection to preserve diversity across required tools. We optimize the policy with GRPO (Shao et al., 2024) using the following hyperparameters: learning rate 1e-6, train batch size 8, rollouts per prompt 8, training steps 100, rollout temperature 0.7, maximum response length 64K tokens.

Across both starting points, our RL training improves evaluation performance over the corresponding baseline. The gain on the vanilla 4B model shows that RL training can improve Claw-style agents even without prior SFT, while the improvement on ClawGym-30B-A3B indicates that the same recipe remains useful after supervised agent training. These preliminary results suggest that lightweight sandbox-parallel rollouts may provide a practical RL path for further improving Claw-style agents. The training curves are shown in

Category	Atomic Actions
Web Information Retrieval	Search engine queries; webpage content extraction; remote resource download; multi-source information verification.
File and Content Handling	Local file reading and content extraction; structured result writing; document editing and rewriting.
Scheduling and Automation	Scheduled or periodic task management; conditional follow-up action triggering.
System and Environment Operations	System status and environment inspection; process monitoring; file, directory, and disk-usage inspection; package installation or updates; recent log inspection.
Data Extraction and Analysis	Structured data extraction from files or web content; record filtering, sorting, and ranking; basic statistics and aggregate computation.
Communication and Coordination	Email or message drafting; tone, clarity, or brevity rewriting; meeting-note or action-item preparation; updates, reports, and status-summary composition.
Development and Execution	Script or command-line execution; code or configuration inspection; code or configuration modification; test or validation execution; command-output and error-message analysis.

Table 9: Atomic action taxonomy used for persona-driven task synthesis.

Category	Subcategories
Office, Communication, and Collaboration	Document writing and editing; meeting notes and summaries; table and record handling; schedule, task, and reminder management; message and email handling.
Content and External Communication	General content writing; marketing content planning; social content creation; SEO and page optimization; email and communication writing.
Information Retrieval and Collection	Information search and lookup; web collection and scraping; research and information synthesis; fact-checking and verification; knowledge retrieval and grounding.
Analysis, Decision Support, and Reasoning	Data analysis and calculation; classification, ranking, and scoring; trend monitoring and detection; mathematical and logical reasoning; algorithmic and technical problem solving.
Automation and Workflows	Scheduled task execution; event-triggered automation; automated inspection and handling; task routing and orchestration.
Software Development	Code generation and implementation; code review and refactoring; test writing and validation; debugging and issue fixing; prototype development and delivery.
Systems and Infrastructure	System status inspection; environment configuration and deployment; DevOps and continuous delivery; incident response and reliability; device and system control.
Planning and Knowledge Organization	Task planning and decomposition; information synthesis and summarization; solution and architecture design; curriculum and content design; memory and knowledge organization.
Professional Domain Support	Legal and compliance review; medical and health analysis; financial analysis and support; security and risk assessment.

Table 10: Scenario taxonomy used for persona-driven task synthesis, covering 9 categories and 43 subcategories.

Category	Description	#Tasks
Productivity and Collaboration	Routine office work, messaging, scheduling, coordination, and collaborative workflows.	44
Systems and Automation	System inspection, workflow automation, operational execution, and environment-level handling.	42
Analysis and Reasoning	Data analysis, evaluation, decision support, problem solving, and technical reasoning.	35
Content and Domain Support	Professional-domain assistance, external-facing writing, reporting, and communication tasks.	28
Planning and Knowledge	Information collection, knowledge organization, retrieval, synthesis, and planning-oriented work.	26
Software Development	Code implementation, debugging, testing, refactoring, and software engineering workflows.	25
All		**200**

Table 11: Category distribution of ClawGym-Bench. The six evaluation categories are grouped from the fine-grained synthesis taxonomy for stable category-wise reporting.

Figure 5.

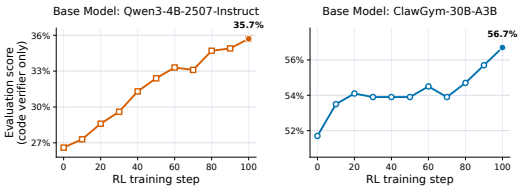


Figure 5: RL training curves on ClawGym-Bench. Scores are computed using only code-based verifiers with weight 1.0, without rubric-based judgment.

D Detailed Behavioral Analysis of Claw Agents

Claw-style tasks require agents to use local files and tools, process execution feedback, and update persistent workspaces. They involve interdependent abilities such as tool-use appropriateness, long-horizon execution robustness, and fine-grained instruction following. We compare GPT-5.4 and Qwen3-30B-A3B (30A3) on ClawGym-Bench cases to identify effective execution patterns and common failures, offering insights for future data synthesis, training, and evaluation.

Tool-Use Appropriateness. In Claw-style environments, task success depends not only on invoking tools, but also on using them coherently within an execution workflow. Figure 6 illustrates this distinction in a CI artifact audit task. The stronger trajectory expands the file pattern, inspects the JSON schema, and runs a Python aggregation script to compute the reports, forming a discovery-inspection-computation-verification pipeline. The weaker trajectory recovers from an invalid wildcard read by using `find` and reading the matched files, but does not turn this recovery into a reliable computation process. Consequently, its outputs remain brittle in summary fields, group counts, filtering semantics, and critical-instance checks. This case shows that tool-use appropriateness goes beyond selecting valid tools: reliable agents must compose exploration, computation, feedback, and verification toward the final objective.

Long-Horizon Execution Robustness. Long-horizon Claw tasks require agents to coordinate many dependent steps, including reading files, running scripts, generating artifacts, checking intermediate outputs, and preserving workspace state across reruns. In such settings, robustness is not defined by whether every action is correct from the beginning, but by whether the agent can interpret feedback, recover from disruptions, and continue along a valid solution path without losing task context. This is important because early failures often affect later artifacts and become visible in the final workspace state. Figure 7 illustrates this behavior in a support-ticket automation task. GPT-5.4 encounters a missing optional memory file, treats it as non-blocking, and proceeds to inspect the inbox files. When a cleanup command is rejected by the execution guard, it avoids repeating the failing action, resets the relevant state file, reruns the processor, and verifies that the processed-batch state contains the two expected batches without duplication. In contrast, 30A3 accumulates unresolved errors and reaches an approval-related dead end, leaving the required rewrites, metrics summaries, notifi-

Case 1: Tool-Use Appropriateness

Task requirement. Generate two CI artifact audit reports from all JSON exports under `input/ci-artifacts/*.json`. The agent must group artifacts by repository and artifact name, compute size statistics, count missing sizes and critical instances, normalize repository identifiers, and write `output/audit/report.json` and `output/audit/report_filtered.json`. The filtered report must use the regex `(coverage|test-results)`.

Strong Model: Reliable Tool Workflow

The higher-scoring trajectory uses tools as a staged computation pipeline. It first expands the file pattern, then inspects the JSON structure, and finally runs a Python aggregation script to compute both reports from the underlying records. The printed summaries expose key quantities required by the verifier.

```
exec: list input/ci-artifacts/*.json
output: gadgets.json, gizmos.json,
  widgets.json
exec: inspect JSON shape with python3
output: keys = repository,
  total_count, artifacts
output: LISTKEY artifacts len 3 / 5
exec: run Python aggregation script
write output/audit/report.json
write output/audit/report_filtered.json
FULL summary:
  files_scanned = 3
  records_scanned = 11
  records_missing_size = 1
  groups = 7
  critical_instances = 2
FILTERED summary:
  artifact_match = (coverage|test-results)
  groups = 3
  critical_instances = 0
reward = 1.000
```

Weak Model: Brittle Tool Use

The weaker trajectory is not a no-op. After an invalid wildcard read fails, it recovers by using `find` and reads the three JSON files. However, the later workflow does not establish a comparably reliable aggregation-and-verification process, leading to only partial satisfaction of the exact report checks.

```
read input/ci-artifacts/*.json
Tool error: ENOENT,
  wildcard not expanded
exec: find input/ci-artifacts
  -name '*.json' | sort
output: gadgets.json, gizmos.json,
  widgets.json
read input/ci-artifacts/gadgets.json
read input/ci-artifacts/gizmos.json
read input/ci-artifacts/widgets.json
...
write output/audit/report.json
write output/audit/report_filtered.json
Unresolved issues:
  exact summary fields
  group ordering / counts
  filtered report semantics
  critical-instance checks
reward = 0.308
```

Figure 6: The stronger trajectory builds a computation-and-verification pipeline, while the weaker one remains brittle in report aggregation despite recovering from an initial tool error.

cations, and idempotent state incomplete. This case shows that long-horizon robustness depends on maintaining coherent progress under execution feedback, so that local disruptions are corrected before they compound into incomplete task execution.

Fine-Grained Instruction Following. Claw-style tasks often contain small but consequential constraints, including filtering rules, output schemas, numeric formulas, and cross-file consistency requirements. Because agents must produce persistent artifacts such as CSV, JSON, Markdown, or scripts, merely creating the requested files does not guarantee task correctness; the artifacts must also satisfy the detailed conditions implied by the instruction and input resources. Figure 8 shows this issue in a reorder-plan task. GPT-5.4 first applies the required condition `Quantity <= ReorderPoint` and then computes downstream supplier outputs from the filtered records. In contrast, 30A3 generates plausible CSV and JSON files but includes invalid items whose quantities exceed their reorder points. These filtering errors propagate into supplier totals and order files, causing the final workspace to appear complete while violating core requirements. This case shows that reliable Claw agents must preserve detailed constraints across generated artifacts and derived outputs, motivating finer-grained supervision over intermediate decisions, artifact contents, and cross-file consistency.

Case 2: Long-Horizon Execution Robustness

Task requirement. Build a local automation for support-ticket batches under `input/inbox/*.jsonl`. For each batch, the agent must generate ticket rewrites, batch metrics, team notifications, and an idempotent state file at `out/state/processed_batches.json`. The automation must process the two provided examples and remain safe to rerun. Here, `batch-04-15` and `batch-04-16` abbreviate the two provided support-ticket input files.

Strong Model: Recovery and Closure

The stronger trajectory encounters errors, but treats them as recoverable execution feedback. After missing optional memory files and later hitting an `exec` preflight refusal, it switches to a safer workflow, resets only the state file, reruns the processor, and verifies that the second run does not duplicate work.

```
read MEMORY.md
Recoverable error: ENOENT
list input/inbox
read batch-04-15.jsonl
read batch-04-16.jsonl
write process_support_batches.py
exec: python3 process_support_batches.py
check generated rewrites/metrics/notifications
edit: preserve API, UTC, README,
  CONTRIBUTING, CSV, CLI
exec: complex cleanup command
Recoverable error: preflight refused
fallback: write state = []
exec: python3 process_support_batches.py
exec: python3 process_support_batches.py
read processed_batches.json
output: two batches, no duplicates
reward = 1.000
```

Weak Model: Failure Cascades

The weaker trajectory does not convert tool failures into a stable recovery plan. After repeated missing-file errors and an approval-related dead-end, the workflow fails to converge to a complete runnable automation. As a result, the required per-batch artifacts and idempotent state tracking are largely missing.

```
attempt to inspect workspace files
Unresolved error: repeated missing files
...
attempt partial setup actions
...
Dead-end: approval request blocks progress
/approve ... allow-once
...
Missing: complete rewrites
Missing: metrics summaries
Missing: team notifications
Missing: valid idempotent state
reward = 0.067
```

Figure 7: A representative case of error recovery in long-horizon execution. The stronger trajectory recovers from tool failures and completes an idempotent workflow, while the weaker trajectory accumulates unresolved errors and fails to reach stable completion.

E Prompt

E.1 Task Synthesis

Persona-Driven Top-Down Synthesis

```
You are designing one realistic computer
    task.

Goal:
Generate exactly one realistic end-user
    request that another coding agent could
    complete on a personal computer.

Given:
- Task category: {task_category}
- Persona: {personal}
- Required basic operations: {
    basic_operations}

Design priorities:
1. Keep the task realistic, concrete, useful
    , and plausible for this persona.
2. Make all required basic operations
    essential steps.
3. Make deliverables, inputs, outputs, and
    key constraints clear but not over-
    specified.
4. Prefer tasks with stable verification
    paths while maintaining naturalness.

Core task rules:
- The task must be self-contained and
    executable in a normal local workspace
    using files, scripts, shell commands, or
     code.
- Do not require private infrastructure,
    internal systems, enterprise accounts,
    remote desktops, special hardware,
    mobile devices, or offline human actions
    .
- The task must produce at least one
    observable artifact or workspace state
    change, preferably with explicit output
    paths under the workspace.
- If input_files are included, every file
    must be necessary, explicitly used, and
    should support cross-validation of
    outputs via recomputation, comparison,
    filtering, aggregation, extraction, or
```

Case 3: Fine-Grained Instruction Following

Task requirement. Generate a reorder plan from `input/inventory.csv` and `input/bulk_discounts.csv`. The reorder list must include only rows where `Quantity <= ReorderPoint`, compute `TargetStock`, `OrderQty`, and `LineTotal` exactly, aggregate supplier orders, and create one JSON file per supplier under `output/orders/`.

Strong Model: Rule-Consistent Generation

The stronger trajectory computes the outputs from the input tables and verifies the generated artifacts. It filters rows by the stated reorder condition, aggregates supplier totals, and creates the complete supplier-level JSON set.

```
read input/inventory.csv
read input/bulk_discounts.csv
exec: compute reorder plan from CSVs
  keep only Quantity <= ReorderPoint
  apply supplier discount rules
write output/reorder_list.csv
write output/supplier_orders.csv
write output/orders/*.json
orders files:
  Acme_Office.json
  CleanWorks.json
  Global_Stationery.json
  Paper_Plus_Co.json
reward = 1.000
```

Weak Model: Plausible but Flawed Output

The weaker trajectory writes structured CSV and JSON files, so the result appears reasonable at a glance. However, it fails to follow the core filtering rule: `output/reorder_list.csv` should contain only items whose current quantity is not above the reorder point, but several included rows violate this condition.

```
write output/reorder_list.csv
...
Required rule:
  include row only if
  Quantity <= ReorderPoint
Actual output violates rule:
  Presentation Folders: 12 > 10
  Correction Tape: 9 > 7
  Disinfectant Wipes: 8 > 5
Why this is wrong:
  these items already exceed their
  reorder points, so they should not
  appear in the reorder list
write output/supplier_orders.csv
Propagated error:
  supplier totals include invalid rows
write output/orders/*.json
Propagated error:
  per-supplier order files inherit
  the invalid reorder items
reward = 0.429
```

Figure 8: A representative case of fine-grained requirement satisfaction. The weaker trajectory produces plausible reorder artifacts but violates the required `Quantity <= ReorderPoint` filtering rule, causing invalid rows to propagate into downstream supplier outputs.

```
    consistency checks.
- Keep the request natural and user-like
    rather than benchmark-like.
- Include only the minimal set of explicit
    requirements needed to make the task
    unambiguous and verifiable.
- Do not over-specify low-level formatting
    details (e.g., exact whitespace,
    indentation, newline rules), tool
    choices, or step-by-step implementation
    unless essential.
- Prefer multi-step reasoning, coordination
    across files, transformations, or non-
    trivial synthesis over simple one-step
    edits.
```

```
- Avoid vague instructions such as "make it
    better" or "handle properly", but also
    avoid turning the task into a checklist
    or specification document.
- Every requirement should correspond to a
    meaningful artifact, output, or
    constraint-not internal implementation
    details.
- Avoid subjective taste, unavailable
    websites, credentials, or unstable
    external state as the core dependency of
     success.

Verification-friendliness rules:
- Prefer tasks whose correctness can be
    verified from deterministic local
```

```
    workspace artifacts rather than unstable
     external web content.
- Do not generate tasks whose validation
    would require hardcoding exact search
    results, page text, downloaded content,
    URLs, or other unstable external facts.
- If a task involves searching or
    downloading, ground success primarily in
     stable local deliverables created after
     that step.
- Avoid tasks where correctness depends on a
     single externally sourced answer that
    the evaluator would need to hardcode.
- When multiple task ideas are possible,
    prefer the one with the most stable and
    least hardcoded verification path.
- Prefer local input_files over live web
    retrieval whenever the same skill can be
     exercised using provided files.

Internet-specific rules:
- Apply these rules only if a required basic
     operation is one of:
  - "perform search engine queries from the
    Internet"
  - "download and extract webpage content
    from the Internet"
  - "download remote resources from the
    Internet"
- In those cases, the task must genuinely
    require live Internet use rather than a
    local substitute.
- input_files may be included only if they
    are strictly necessary, do not reveal
    the target external information, and do
    not make the Internet step unnecessary.
- If both Internet data and input_files are
    used, the task should combine them in a
    way that supports clear, stable, locally
     verifiable outputs.
- Do not give a direct URL; identify the
    target resource by objective attributes
    such as official source, organization,
    document title, file name, publication
    identifier, or domain pattern.
- Do not rely on latest, current, recent,
    trending, search ranking, mirror choice,
     or fragile page structure as part of
    correctness.
- Do not make exact external facts, rankings
    , snippets, article wording, or
    downloaded file contents the primary
    grading target.
- Require external information to be
    transformed into structured local
    artifacts such as tables, JSON, CSV, or
    reports with fixed sections or fields.
- Prefer tasks where searched or downloaded
    content is only one input to a larger
    local processing workflow.
- The Internet step must remain essential.
    If the task could be completed correctly
     using only input_files without Internet
     access, the task design is invalid.

Output requirements:
- The question must be written in {
    question_language}.
```

```
- Output exactly one JSON object and nothing
     else.
- Use this shape:

  {{
    "question": "string",
    "task_type": "string",
    "scenario": "string",
    "expected_behavior": "string",
    "required_basic_operations": ["string"],
    "has_input_files": true_or_false,
    "difficulty": {{
      "score": 1,
      "rubric": "1=single-step/light edit;
    2=simple multi-step; 3=moderate
    reasoning/coordination; 4=hard multi-
    step with tight constraints; 5=very hard
     multi-stage synthesis/debugging"
    }}
  }}

- Include "input_files" only when
    has_input_files is true.
- When included, input_files must be a list
    of objects with:
  - "file_path": relative path such as "
    input/file.ext"
  - "file_format": one of "txt|csv|json|
    jsonl|md|tsv|yaml|xml|html|py"
  - "content": full file content as a string

Field guidance:
- question:
  - Write the actual user request shown to
    the coding agent.
  - Mention required input and output paths
    explicitly.
  - Define enough constraints to make the
    task unambiguous and verifiable, but
    avoid unnecessary low-level detail.
  - Specify output formats, fields, or
    structure when they materially affect
    correctness.
  - If input_files are provided, include
    requirements that allow cross-validation
     against them.
  - Avoid unclear pronouns or implicit
    expectations.
  - If Internet use is required, do not
    include a direct URL.
- task_type:
  - Use a concise label aligned with the
    task category.
- scenario:
  - Briefly explain why this persona would
    ask for the task.
- expected_behavior:
  - Summarize the intended successful
    outcome at a high level.
- required_basic_operations:
  - Include all required operations from the
     input.
- has_input_files:
  - Set based on your design.
  - If Internet is required, ensure
    input_files do not replace it.
- difficulty:
  - score must be 1-5.
```

```
  - rubric must match the provided scale
    exactly.

Return JSON only.
```

Skill-Grounded Bottom-Up Synthesis

```
You are generating realistic task entries
    for a dataset.

Runtime path convention for this generation
    step:
- When the task mentions runtime paths,
    always use relative workspace paths.
- Use input/ for agent-readable inputs.
- Use output/ for agent-created outputs.
- Use reward/ only when you must refer to
    reward-side files.
- Do not use placeholders or hardcode
    machine-specific absolute paths.

Use the annotations below as the ground
    truth for what the available skills can
    do. Generate exactly {task_count}
    distinct tasks.

Task synthesis source mode:
- {task_source}
- "original" means the original concatenated
     skill content
- "core_content" means the abstracted core-
    content seed extracted during stage 1
- Use the selected task-source content below
     as the main ideation seed for this
    skill, while treating the structured
    annotation as the ground-truth
    capability and constraint summary.

Requirements:
{validation_mode_task_guidance(
    validation_mode)}
- The primary skill must be essential to
    every task.
- If supporting skills are provided, every
    generated task must genuinely require
    all listed supporting skills as part of
    the solution.
- Every task must avoid visual and audio
    media.
- If a task uses files, every task-relevant
    input/output file must use one of these
    extensions only: .txt, .csv, .json, .
    jsonl, .md, .tsv, .yaml, .xml, .html, .
    py.
- Do not generate any task that requires or
    produces images, screenshots, video,
    audio, or other visual/audio media files
    .
- Do not expose internal labels like "
    primary skill" or "supporting skill" in
    user_query.
- All natural-language task content must be
    in English only. Do not use Chinese or
    any other non-English language anywhere
    in user_query, reward_summary,
    difficulty_reason, input_files, or
    expected_behaviors.
- Do not mention or require environment-
    specific dependencies or implementation
    details such as Skills, MCPs, tools,
    plugins, SDKs, APIs, libraries, packages
    , CLIs, shells, runtimes, or the
    underlying skill names.
- user_query must contain a concrete,
    actionable request with a clear
    deliverable or outcome. Do not output
    vague background descriptions without an
     explicit task to complete.
- Every task must feel like a real user
    request, not a benchmark spec.
- Tasks must be complex and realistic. They
    should require multiple steps, multiple
    capabilities, and multiple turns of
    agent interaction to complete.
- Prefer difficulty 3, 4, or 5 unless the
    skill genuinely cannot support that
    complexity.
- user_query must be the exact first user
    message sent to the agent.
- If input files are needed, user_query must
     explicitly tell the agent to read files
     under input/.
- If the agent should create files,
    user_query must explicitly tell the
    agent to write them under output/ with
    concrete paths.
- input_files should be a brief list of
    required input files, including filename
     and purpose, for example "question.txt
    - algebra problem statement".
- Every input_files entry must name a file
    whose extension is one of: txt, csv,
    json, jsonl, md, tsv, yaml, xml, html,
    py.
- reward_type must be exactly "output_files".

- reward_summary must clearly describe what
    the reward script should inspect and how
     success should be judged.
- reward_summary must describe only file-
    based checks against artifacts under
    output/. Do not mention final_message or
     the assistant response.
- reward_summary should also use relative
    runtime paths when referring to files.
{validation_mode_task_evaluation_fit(
    validation_mode)}
- Output valid JSON only.
- Output a JSON array only, with no prose
    and no markdown fences.

Return exactly this schema for each item:
[
  {
    "user_query": "string",
    "requires_input_files": true,
    "input_files": ["string"],
    "reward_type": "output_files",
    "reward_summary": "string",
    "expected_behaviors": ["string"],
    "difficulty": 4,
    "difficulty_reason": "string"
  }
]
```

```
Skill ID: {skill_id}

Primary skill annotation:
{annotation_json}

Primary task-source content:
{task_source_content}

Supporting skill references (annotation +
    task-source content):
{supporting_annotations_json}
```

Skill Annotation

```
You are analyzing one OpenClaw skill so a
    downstream pipeline can synthesize
    realistic training and evaluation tasks
    from it.

Read the full skill content carefully before
     answering. Your job is to summarize the
     capability, identify whether it needs
    real authentication or credentials,
    describe the inputs and outputs, note
    any meaningful risks, and list external
    runtime dependencies.

Important rules:
- Output valid JSON only.
- Do not wrap the JSON in markdown fences.
- Do not include any explanation before or
    after the JSON.
- If auth_required is true, auth_details
    must be a non-empty list.
- If auth_required is false, auth_details
    must be [].
- If has_risk is true, risk_reasons must be
    a non-empty list.
- If has_risk is false, risk_reasons must be
     [].
- If text_only_compatible is true,
    non_text_reasons must be [].
- If text_only_compatible is false,
    non_text_reasons must be a non-empty
    list.
- language must be one of: "english", "
    chinese", "multilingual", or "other".
- core_content must be a non-empty string.
- Mark auth_required as true when the skill
    requires a real account, login, API key,
     OAuth token, secret, or any other
    credential that cannot be reasonably
    mocked for synthetic task generation.
- Mark text_only_compatible as false when
    the skill's core capability or realistic
     tasks inherently require visual or
    audio media, such as images, screenshots
    , video, or audio.
- Classify language by the primary human
    language used in the skill content and
    examples. Use "multilingual" only when
    multiple human languages are materially
    used.
- Treat only these file types as allowed for
     downstream synthesized tasks: txt, csv,
     json, jsonl, md, tsv, yaml, xml, html,
    py.
- core_content should be an abstract,
    reusable seed for downstream task
    synthesis. It should capture the skill's
     essential capability, common workflows,
     important constraints, typical inputs
    and outputs, and reusable patterns,
    while reducing unnecessary
    implementation-specific detail.
- core_content should be concrete enough to
    support diverse task synthesis, but
    abstract enough to generalize beyond the
     exact wording or incidental specifics
    of the original skill file.
- Be concrete and concise. Avoid vague
    generic wording.

Return exactly one JSON object with this
    schema:
{{
  "language": "english",
  "summary": "string",
  "auth_required": true,
  "auth_details": ["string"],
  "input_format": "string",
  "output_format": "string",
  "has_risk": true,
  "risk_reasons": ["string"],
  "required_env": ["string"],
  "text_only_compatible": true,
  "non_text_reasons": ["string"],
  "core_content": "string"
}}

Skill ID: {skill_id}

Full skill content:
{skill_content}
```

E.2 Verification Generation

Verification Code Generation

```
You are a senior verification-grading
    engineer.

Your job is to generate executable Python
    grading code from a task request and
    optional input files when no
    verification points are available.

The main objective is to output a complete,
    well-formed, executable grader script in
     a single fenced Python code block.

Critical parser warning:
- The downstream pipeline extracts Python
    code from a fenced code block.
- Therefore do not output `<think>`, prose,
    JSON, or any text outside the code block
    .
- If you output anything before or after the
     code block, extraction may fail.

Return exactly one fenced Python code block
    and nothing else:

```python
```
```

```
# complete standalone Python code here
```

Output rules:
- The response must start with ```python
- The response must end with ```
- Do not output analysis, `<think>`, JSON, prose before the code block, or prose after the code block.
- Do not output multiple code blocks.
- The code block content must be a non-empty standalone Python 3 program.

Primary goal:
Generate high-quality executable grading code from only a task request and optional input files. No verification points are provided.

Infer a strict, deterministic grading strategy from the explicit task requirements and any provided input files.

Inference requirements:
- Infer grading logic from the task request and input files only. Do not assume hidden requirements, hidden deliverables, or unstated preferences.
- Derive atomic checks from the explicit task text itself, and cover every major explicit obligation with one or more returned score keys.
- Prefer layered verification over minimal artifact-existence checks.
- If the task specifies exact paths, filenames, sections, fields, ordering, counts, schemas, headings, literals, or workflow behavior, implement equally strict checks.
- When outputs are derived from input files, prefer recomputation and comparison: records, summaries, selections, joins, transformations, rankings, statistics, and cross-file consistency.
- For natural-language outputs, prefer fact-grounded, structure-aware, and consistency-aware checks over whole-body string equality unless exact wording is explicitly required.
- If no input files are provided, infer the strongest deterministic checks you can from the task request and observable workspace state without inventing new requirements.

Implementation requirements for the generated code:
- You are writing the grader, not solving the task.
- Use Python 3 standard library only.
- Define exactly `def grade(transcript: list, workspace_path: str) -> dict:`.
- Return floats in `[0.0, 1.0]`, not booleans.
- Use `Path(workspace_path)` for workspace access.
- Treat `transcript` as a required function parameter only; do not use transcript-based checks for grading.
- Prefer checking workspace files and other observable workspace artifacts directly.
- Use `subprocess` only when the task explicitly depends on command outcomes or installed packages, and keep it deterministic.
- The grader should not depend on network access.
- The grader should not modify workspace files as part of grading.
- Handle missing files and malformed JSON/CSV/text gracefully by returning `0.0` for affected checks.
- The script must be independently executable as a full Python file.
- Include a minimal CLI entrypoint that accepts the workspace path as the first positional argument, for example `python generated_validation.py /path/to/workspace`, and prints only the JSON grade result.
- Do not require a named flag such as `--workspace` for the workspace path. If no positional argument is provided, default to `"."`.
- Make helper functions when they improve clarity.
- Do not create score keys whose only purpose is to confirm pre-existing input files, workspace setup, or other supplied environment state.
- File existence, openability, and parseability may be baseline gates, but they should not be the core grading signal unless the task is only about creating that artifact.
- Prefer structural checks for organization requirements and objective content checks when expected results can be recomputed or directly compared.
- If the task text is underspecified in one area, grade only what is explicit or a direct necessary consequence of an explicit requirement.

Additional robustness requirements:
- The grader must work in both an empty workspace and a minimally populated workspace where expected files exist.
- For every major check, handle both missing-file and present-file branches without crashing.
- If a required file is malformed, partially invalid, or fails to parse, fail the affected check instead of skipping bad rows, bad fields, or bad values.
- Prefer strict checks over permissive substring checks when the task requires exact paths, exact ordering, exact counts, exact line structure, or exact record structure.
- Do not validate only a subset of rows or items when the task requires all rows or items to satisfy the condition.

Code shape requirements:
- The script should be organized in this order:
  1. imports
  2. small helper functions
  3. `grade(transcript: list, workspace_path: str) -> dict`
  4. `main()`
  5. `if __name__ == "__main__":`
- Prefer `from pathlib import Path`.
- Prefer helper functions such as safe file reading, safe JSON loading, and safe CSV parsing when useful.
- Inside `grade(...)`, create a `scores` dict and populate every returned key explicitly.
- If a prerequisite file is missing, do not crash and do not omit keys; return `0.0` for impacted checks.
- Keep test-point names stable snake_case.
- Keep checks atomic when possible instead of one giant combined score.
- When the task has multiple distinct obligations, use multiple score keys rather than only one or two broad checks.
- Do not include placeholder code, pseudo-code, TODOs, or comments like "implement this".
- Do not emit example code fragments; emit the final complete script.
- Ensure helper function signatures match all call sites.

Target code shape:

```python
import json
from pathlib import Path


def _helper(...):
    ...


def grade(transcript: list, workspace_path: str) -> dict:
    workspace = Path(workspace_path)
    scores = {
        "example_check": 0.0,
    }
    ...
    return scores


def main() -> None:
    ...


if __name__ == "__main__":
    main()
```

Important:
- This is a shape guide, not literal code to copy.
- Replace `"example_check"` with real stable snake_case score keys.
- The final script must be complete and executable, not a sketch.

Generation instructions:
- Treat the task request as the source of truth.
- Infer a rigorous verification plan from the explicit task requirements and any provided input files.
- Do not invent deliverables, constraints, or hidden expectations not supported by the task text.
- Cover every major explicit obligation with one or more score keys.
- Prefer complementary checks: artifact validity, structure, source-derived correctness, cross-file consistency, and constraint compliance when the task supports them.
- If the task names exact paths, headings, fields, counts, formulas, ordering rules, or exact wording, implement equally strict checks.
- For flexible text outputs, prefer fact-grounded and structure-aware checks unless exact wording is explicitly required.
- If no input files are provided, derive the strongest deterministic checks you can from the task request and observable workspace artifacts.
- The output code should look like production-ready benchmark code, not a chat response.
- Do not output metadata such as test_points, notes, or coverage maps.

Use the following task materials as the source of truth:

{task_materials}

Return only one ```python ... ``` block.

Rubric Verification Generation

You are generating an LLM rubric for one OpenClaw synthetic task.

The task question below is the source of truth. Produce valid JSON only. Do not add markdown fences or any extra text.
You are also given executable verification code for the same task.

Your job is to generate rubric rules ONLY for important quality dimensions that the verification code does NOT already reliably check.

Hard requirements:
- eval_type must be "llm_rubric".
- score_options must be exactly [0, 0.25, 0.5, 0.75, 1].
- rules may contain one or more items depending on the task, but keep them minimal and necessary.

```
- rules must be specific, concrete, and non-
    overlapping enough to support reliable
    scoring.
- Each rule must define concrete
    descriptions for score levels 0, 0.25,
    0.5, 0.75, and 1.
- Each rule must include a file_path field
    naming the primary output file being
    judged.
- eval_input should explain what material is
     shown to the evaluator.

Critical rubric boundaries:
- This rubric is ONLY for subjective or non-
    fully-code-verifiable aspects of the
    answer.
- Treat the provided verification code as
    already covering objective executable
    checks.
- Do NOT create rubric rules for anything
    the verification code already checks
    directly or via deterministic
    recomputation/pattern matching.
- Do NOT restate task checklist items that
    are already objectively covered by code.
- Do NOT spend rubric rules on artifact
    existence, parseability, exact filenames
    , exact counts, exact field names, exact
     section titles, exact phrases, exact
    links, exact schemas, exact ordering,
    exact numeric correctness, or other code
    -verifiable basics unless the code
    clearly does not cover them and the task
     still needs subjective review there.
- Only include rules for quality dimensions
    that are not already reliably covered by
     the verification code.
- Even if the current verification code does
     not cover a dimension, do NOT generate
    a rubric rule for it if that dimension
    could be checked reliably with simple
    deterministic code and does not require
    meaningful human judgment.
- If no meaningful subjective quality
    dimension remains after applying these
    constraints, do NOT generate a rubric.
- In that case, return exactly this JSON
    instead of a rubric:
{{
  "skip_rubric": true,
  "reason": "short string"
}}
- Use this skip output only when any
    additional rubric rule would be
    redundant with the verification code,
    almost entirely objective, or too weak
    to justify rubric-based evaluation.

Task-faithfulness requirements:
- The task question is the source of truth.
    Do NOT introduce new requirements beyond
     the task.
- Do NOT turn "nice-to-have" qualities into
    rubric requirements unless they are
    explicitly required or directly implied
    by the task.
- Do NOT add stronger expectations than the
    task states, such as extra formatting
    polish, deeper analysis, stricter
    brevity, more formal tone, more detailed
     structure, or additional sections,
    unless the task explicitly requires them
    .
- If the task clearly specifies or strongly
    implies a persona, audience, voice, tone
    , or stylistic frame, you MAY include
    rubric rules that judge how well the
    output fits that frame.
- Only do this when the persona/style
    requirement is explicit or directly
    implied by the task; do NOT invent a
    persona or stylistic requirement that is
     not grounded in the task.
- If the task is almost fully code-
    verifiable, keep the rubric very light
    or return only a very small number of
    rules.

What rubric rules SHOULD focus on:
- Prefer complementary quality dimensions
    the code is unlikely to judge reliably,
    such as explanation clarity,
    interpretation quality, groundedness in
    source material, comparison
    distinctiveness, actionable usefulness,
    prioritization quality, reader usability
    , tone appropriateness, or non-
    misleading framing.
- When relevant to the task, persona fit,
    audience fit, voice consistency, tone
    appropriateness, and stylistic alignment
     are valid rubric dimensions.
- Focus on qualities of the assistant's
    produced output, not on the setup
    metadata around the task.
- Anchor every rule to a concrete output
    file path mentioned or implied by the
    task.
- The rule descriptions must be specific to
    the expected content of that file, not
    generic comments like "good quality", "
    reasonable output", or "clear writing".
- Multiple rules may point to the same
    file_path when that file needs judgment
    on multiple distinct dimensions.
- Each rule should target one main dimension
     only. Avoid bundling multiple unrelated
     criteria into one rule.
- Keep the rules concrete enough that a
    reviewer can distinguish partial success
     from strong success.

Scoring language requirements:
- Make the 0 / 0.25 / 0.5 / 0.75 / 1
    descriptions meaningfully different and
    easy to distinguish.
- Avoid vague adjectives like "good", "
    strong", "professional", "polished", or
    "clear" unless tied to specific
    observable qualities.
- Prefer concrete failure/success
    descriptions over abstract praise.

Return exactly this JSON schema:
{{
  "eval_type": "llm_rubric",
```

```
  "score_options": [0, 0.25, 0.5, 0.75, 1],
  "eval_input": "string",
  "rules": [
    {{
      "name": "string",
      "file_path": "string",
      "scores": {{
        "0": "string",
        "0.25": "string",
        "0.5": "string",
        "0.75": "string",
        "1": "string"
      }}
    }}
  ]
}}

Or, if no rubric should be generated, return
      exactly:
{{
  "skip_rubric": true,
  "reason": "short string"
}}

Task question:
{task_question}

Verification code:
```python
{validation_code}
```
"""
```

E.3 Quality Assessment

Task Reasonableness Assessment

```
You are a senior evaluator for computer-task
      dataset quality. Your job is to
     carefully review one task specification
     and judge whether it is factually sound
     and whether it leaks the answer.

You are not solving the task. You are
     auditing the task itself. Be
     conservative, precise, and literal. Base
      your judgment on the task text and the
     provided metadata only.

Your job is to evaluate the task on exactly
     two dimensions:
1. Whether the task contains a factual error
      that makes it not realistically
     solvable as written.
2. Whether the task leaks part or all of the
      answer.

Return JSON only, with exactly this schema:
{
  "has_factual_error": 0,
  "has_factual_error_reason": "",
  "has_answer_leak": 0,
  "has_answer_leak_reason": ""
}

Field rules:
- has_factual_error: 1 if the task contains
     a factual mistake, impossible
     requirement, or contradictory setup such
      that the task cannot realistically be
     completed as written; otherwise 0.
- has_answer_leak: 1 if the task gives away
     part or all of the desired final answer/
     output in a way that substantially
     reduces the intended solving work;
     otherwise 0.
- Each *_reason field must be a concise
     explanation grounded in the task text.
- Do not output markdown, code fences, or
     extra keys.

Detailed judging guidance:
- Focus on the task as written, not on
     hypothetical fixes or charitable
     reinterpretations.
- Do not assume missing details will be
     provided later.
- Judge from the perspective of whether
     another coding agent could complete the
     task correctly as written.

How to judge factual_error:
- Set has_factual_error = 1 when the task
     includes an impossible fact, a
     contradiction, a broken dependency on
     unavailable or false information, or
     instructions that cannot all be
     satisfied together.
- Set has_factual_error = 1 when the task
     asks for something that is not actually
     achievable as specified, such as using
     the wrong file, wrong data, wrong field,
      or invalid assumptions about the
     environment.
- Set has_factual_error = 0 when the task is
      merely difficult, underspecified in a
     minor way, or could reasonably be
     completed despite small ambiguity.

How to judge answer_leak:
- Set has_answer_leak = 1 when the task
     itself reveals part or all of the
     intended final answer/output/content so
     that the intended reasoning or
     generation work is substantially already
      done.
- Examples of answer leakage include: giving
      the completed text that only needs to
     be copied, listing the exact filled
     values that constitute the main answer,
     embedding the solved output inside the
     task, or otherwise collapsing the task
     into formatting/transcription.
- Set has_answer_leak = 0 when the task
     gives normal constraints, requirements,
     filenames, target fields, or output
     expectations without giving away the
     substantive final answer.

Reason-writing guidance:
- Each reason should be concise but specific.

- Mention the concrete part of the task that
      drove the judgment.
```

```
- Do not hedge excessively.

Output discipline:
- Return JSON only.
- Do not output markdown.
- Do not include any keys beyond the
    required schema.

Task metadata:
- record_index: {record_index}
- task_category: {task_category}
- personal: {personal_json}
- basic_operations: {basic_operations_json}

Task text:
{task_text}
```

Code Quality Assessment

```
You are evaluating a benchmark task prompt
    and its checker code.

Score the pair on exactly two dimensions
    from 0 to 4, where higher is better:

1. checker_coverage_quality
2. checker_alignment_quality

Scoring guide:

For checker_coverage_quality:
- 0 = core requirements largely unverified
- 1 = major important requirements weakly
    checked
- 2 = some important checks exist, but
    notable gaps remain
- 3 = core requirements mostly well verified
- 4 = core requirements strongly and
    appropriately verified

For checker_alignment_quality:
- 0 = checker substantially changes the task
     by enforcing many unstated requirements
- 1 = checker often enforces unstated
    constraints
- 2 = some validations materially exceed the
     task
- 3 = checks mostly align with the task,
    with only small overconstraints
- 4 = checks align closely with the task and
     do not unnecessarily restrict valid
    outputs

Instructions:
- Be conservative, literal, and evidence-
    based.
- Judge benchmark quality, not creativity.
- Do not assume hidden requirements.
- Use the task prompt as the source of truth
     for what the checker is supposed to
    verify.
- Consider both the task and the checker
    when judging coverage and alignment.
- Each score must include a short reason.
- Output valid JSON only.

Return exactly this schema:

{
  "task_summary": "string",
  "scores": {
    "checker_coverage_quality": {
      "score": 0,
      "reason": "string"
    },
    "checker_alignment_quality": {
      "score": 0,
      "reason": "string"
    }
  }
}

Now evaluate the provided task prompt and
    checker code.

Task prompt:
{{TASK}}

Input files:
{{INPUT_FILES}}

Checker code:
```
{{CODE}}
```
```

Overall Check of Code and Rubric

```
You are performing benchmark data quality
    review. Based on the provided
    information, complete the following task
    :

Judge whether the task's validation setup is
     reasonable and good, and also judge
    whether the task itself is a good
    benchmark task.

You need to evaluate two kinds of validation
     separately:
- Code-based validation: from the `content`
    of entries in `input_files` whose `
    file_path` is `reward/check.py` or `
    reward/test.py`.
- Rubric-based validation: from the `rules`
    field. If `rules` is empty, null, or
    missing, do not force an evaluation of
    the rubric; instead, clearly state why
    it was not evaluated.

When judging, focus on:
- Whether the core task requirements are
    covered
- Whether the validation aligns with the
    task requirements rather than
    introducing many extra unstated
    restrictions
- Whether it is clear, executable, and
    stable
- Whether it over-relies on fragile
    formatting, incidental phrasing, or
    unnecessary hard-coding
- When considering code-based validation and
     rubric-based validation together,
```

```
    whether they complement each other, or
    instead overlap heavily, or both leave
    obvious gaps
- Whether the task itself is suitable as a
    benchmark task: whether it is clear,
    verifiable, has well-defined boundaries,
     reasonable difficulty and constraints,
    and can distinguish model ability rather
     than environmental accidents

Requirements:
- Be conservative and evidence-based. Do not
     invent hidden requirements.
- `code_judgment`, `rubric_judgment`, `
    overall_judgment`, and `
    benchmark_task_judgment` should all be
    concise and evidence-based.
- `code_judgment` should evaluate only the
    code-based validation itself.
- `rubric_judgment` should evaluate only the
     rubric-based validation itself.
- `overall_judgment` should evaluate only
    whether the two together are
    complementary, whether there is obvious
    overlap, and whether the overall
    coverage is more complete.
- `benchmark_task_judgment` should evaluate
    only whether the task itself is a good
    benchmark task. Do not focus mainly on
    checker details, though you may
    reference whether the checkers support
    it being a good benchmark.
- All four judgment fields should be as
    concise as possible. Prefer direct
    conclusions and key reasons over long
    explanations; usually 2 to 4 sentences
    are enough.
- If one category is absent, explicitly
    write: "Not provided, so this part
    cannot be evaluated."
- Do not output any scores, ratings, stars,
    boolean fields, list fields, or nested
    scoring structures.
- The output must be valid JSON, and you
    must output JSON only.

The output format must be exactly:
{
  "code_judgment": "string",
  "rubric_judgment": "string",
  "overall_judgment": "string",
  "benchmark_task_judgment": "string"
}

Below is the content to analyze.

Task prompt:
{{TASK}}

Task required input files (these are the
    files the task depends on):
{{TASK_INPUT_FILES}}

Code checker files:
{{CODE_CHECKERS}}

Rubric rules:
{{RUBRIC_RULES}}
```

E.4 Evaluation

Rubric-Based Judge

```
# System prompt:

You are a strict rubric-based evaluator. Use
     only the user prompt content. Do not
    call tools, browse, inspect files, or
    ask for more context. Your response may
    include concise analysis, but it must
    end with exactly one standalone JSON
    object with keys `scores` and `notes`.


# User prompt:

You are grading an OpenClaw agent result.

You must not call, request, or simulate any
    tools. Do not browse, list directories,
    open files, inspect the workspace, or
    ask for more context. Grade only from
    the task, final output files, optional
    transcript evidence, and rubrics
    included in this prompt.

Before giving the final judgement, provide
    concise analysis explaining how the
    final outputs satisfy or fail each
    rubric criterion.

Your final judgement must end with exactly
    one standalone JSON object and nothing
    else after it. The final JSON object
    must contain exactly two keys: `scores`
     and `notes`.

- `scores` must map every rubric id (`
    criterion_1`, `criterion_2`, ...) to
    one numeric score chosen from that
    rubric's allowed score anchors.
- `notes` must be a concise string
    summarizing the main reasons for the
    assigned scores.

Do not include any overall score in the
    final JSON. Do not compute or report the
     final aggregated score. Score
    aggregation will be handled separately
    by post-processing.

## User Task
{{USER_TASK}}

## Final Output Files
{{FINAL_OUTPUT_FILES}}

## Additional Changed Workspace Files
{{ADDITIONAL_CHANGED_WORKSPACE_FILES}}

## Transcript
{{TRANSCRIPT_OPTIONAL}}

## Rubric
{{RUBRIC}}
```