Joint Optimization of Tool Creation and Use for Large Language Model Agents

Zhi Rui Tam[1,2*] **Chieh-Yen Lin**[1] **Yun-Nung (Vivian) Chen**[2]
Shao-Hua Sun[1,2] **Hung-yi Lee**[2]
[1]Appier AI Research, [2]National Taiwan University
ray.eed08g@nctu.edu.tw

Abstract

Tool-augmented language models are bounded by the APIs humans bothered to write; existing tool-creation systems patch this by prompting a frozen LLM at inference time, leaving the model that writes a tool decoupled from the one that uses it, with no signal that the schemas it produces are schemas it can invoke. We propose **SMITH** (Schema-grounded Multi-task Iterative Tool Honing), a reinforcement learning framework that jointly trains tool creation and tool use inside a single policy. Each rollout is either a *build* task (write a tool from a few examples) or a *use* task (invoke a pooled tool on a held-out question). Three separate reward axes catch schema, code, and outcome failures independently, so each failure mode contributes its own gradient. A 4B Qwen3 trained with SMITH on 13 procedural reasoning tasks with exact verifiers reaches 79.8 macro-average accuracy on held-out tasks, the best across all evaluated methods and ahead of an untrained 30B-A3B tool-writer. It also reaches 40.4 on TabMWP-Hard and 42.6 on out-of-domain GQA ($+7.6$ over the best same-backbone inference-time baseline), without any visual or tabular training data. When invoked by a frozen 350M student, tools written by our 4B match those produced by a writer an order of magnitude larger. The same recipe also lifts Qwen3-8B and Granite-3.3-8B without modification. [2]

1 Introduction

Human progress depends on accumulated tools. Rather than solving every problem from first principles, people rely on instruments refined over generations [1, 2]. Large Language Models (LLMs) face a similar limitation: relying only on parametric memory restricts their ability to perform exact computation [3], access up-to-date knowledge [4], and carry out reliable symbolic reasoning [5]. Tool-augmented LLMs were introduced to overcome these limits [6–8]. By invoking external interfaces such as calculators, code interpreters, and search engines, models can solve problems beyond what is encoded in their frozen weights.

However, existing tool-augmented systems still rely on fixed, human-designed toolsets. Such predefined APIs may be incomplete, poorly matched to new tasks, or entirely unavailable, placing a hard limit on model capability. This motivates a shift from static tool use to *dynamic tool creation*, where models synthesize reusable callable functions on demand. LATM [9] first explored this direction by prompting GPT-4 to generate JSON-schema tools from demonstrations, and later work incorporated retrieval, verification, and multi-stage generation pipelines [10–12]. Despite this progress, existing methods do not explicitly optimize for tool quality or reusability during training of LLMs. Moreover, they separate tool creation from tool use: a stronger model writes the tool while a weaker model invokes it [9, 10]. As a result, the tool creator is never incentivized to design schemas it can reliably

*Work done while at Appier AI Research
[2]Source code available at: https://github.com/appier-research/smith

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

use itself. A key open problem is therefore *how to train a single model to become both a better tool creator and a better tool user*.

Reinforcement learning provides a natural framework for this objective [13–15]: a model can create a tool, apply it to held-out queries, and directly optimize against answer correctness. Yet, training under this paradigm introduces two major challenges. First, **reward decomposition**: a generated tool contains both a *schema* (*e.g.*, function name, parameters, and types) and a *backend implementation* (*e.g.*, executable Python code). Failures in these two components require different corrective signals, making reward assignment nontrivial. Second, **circular evaluation**: evaluating tool quality requires a judge, but self-evaluation is inherently unreliable [16], while fixed external evaluators raise questions about trustworthiness and alignment [17]. In addition, a competent evaluator must itself understand tool use well enough to assess whether a schema is practically callable and reusable across harder downstream queries.

To address these challenges, we propose **SMITH** (**Schema-grounded Multi-task Iterative Tool Honing**), a reinforcement learning framework that jointly trains tool creation and tool use within a single policy. SMITH disentangles schema and implementation failures through three complementary reward signals: execution accuracy, LLM-as-judge quality, and format consistency. To reduce circular evaluation, the judge is periodically synchronized from the evolving policy rather than sharing live training weights, enabling the evaluator to improve alongside the policy while maintaining stability. Finally, SMITH evaluates tools created on simpler tasks using harder downstream queries, explicitly rewarding reusable abstractions rather than task-specific shortcuts.

We train SMITH on 13 procedural task families from Reasoning-Gym [18] and evaluate both in-domain generalization and zero-shot transfer to unseen benchmarks. Our 4B model achieves the best overall held-out Reasoning-Gym performance (79.8% macro accuracy), outperforming inference-time tool-writing frameworks such as LATM [9], CRAFT [10], Trove [11], and KTCE [12], as well as models distilled from substantially larger teachers. Despite using far fewer decoding tokens, SMITH also surpasses a 30B inference-time tool writer on unseen tasks. Beyond procedural reasoning, SMITH transfers zero-shot to entirely new modalities: it achieves state-of-the-art performance on TabMWP-Hard and improves GQA visual question answering by up to $+7.6$ points over same-backbone baselines, despite never being trained on tabular or visual data. The learned tools further generalize across models: tools written by our 4B policy enable a frozen 350M model to match the performance of a 30B tool writer. Finally, the same training recipe consistently improves both Qwen3-8B and Granite-3.3-8B, suggesting that jointly optimizing tool creation and tool use under structured rewards is a scalable path toward reusable and transferable tool-building capabilities.

2 Related works

Early tool-augmented LLMs invoke search engines or self-supervised API calls to extend parametric memory [6, 7, 19], but are bounded by a fixed, human-curated toolset.

LATM [9] introduced dynamic tool creation, using a powerful LLM to write reusable tools a weaker model can invoke, yielding stronger performance on BIG-Bench [20]. Subsequent works added increasingly complex scaffolding: CRAFT [10] builds a retrieval-augmented tool library with verification; Trove [11] introduces tool induction and verification pipelines; and KTCE [12] automates creation and evaluation through multi-stage decomposition. Despite their differences, all of these systems treat tool writing as a prompting problem at inference time, leaving tool quality to emerge incidentally from generation rather than from an explicit objective and none couples tool creation with tool use in a joint learning objective.

A parallel line applies RL to code and tool generation: CodeRL [21], RLEF [22] use execution feedback to train better code generators and ToolRL[23] shows tailored design reward can help tool use, while ReTool [24] trains models to invoke tools more reliably without addressing creation. The closest work is SAGE [25], which frames skill creation as an RL objective; however, SAGE uses a single question for creation and a single for validation, limiting generalization across diverse task categories. Our work differs in three key respects: we jointly train tool creation and use across 13 procedural task categories; we validate each generated tool on up to 16 held-out questions; and we apply an LLM judge scoring correctness, schema quality, and overall tool quality as an independent reward signal.

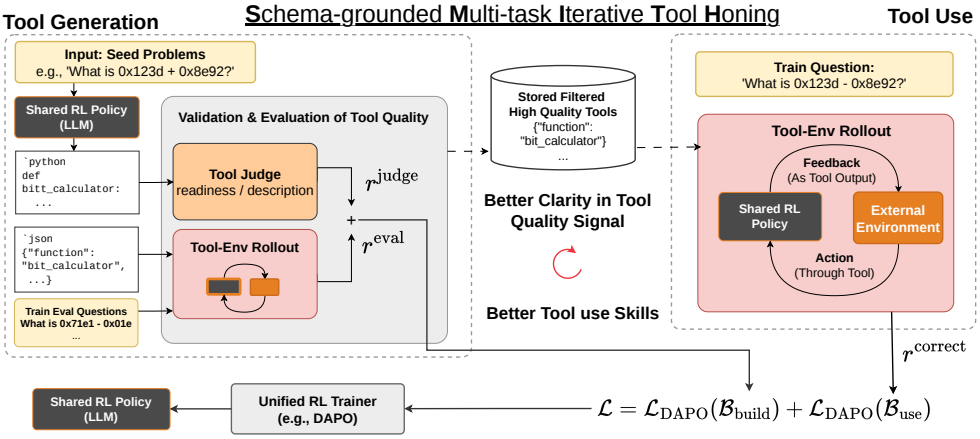

Figure 1: SMITH jointly trains tool creation and tool use in a single shared policy. The build task rewards schema correctness and execution accuracy; the use task rewards answer correctness using only the schema. Because the same model that writes a tool must also invoke it, ambiguous or broken schemas are penalized directly—a feedback loop that prompting-only approaches cannot provide.

SMITH sits at the intersection of program-aided reasoning [26, 27] and program induction [28], inheriting their use of executable code, synthesis of reusable procedures, and utility of compounding tools between tasks. It departs from all three by making these capabilities explicit RL objectives: the *build* task directly rewards concise, interpretable, and reusable tool synthesis, while the *use* task closes the loop by ensuring what the model writes is what it can reliably invoke.

3 SMITH: Schema-grounded multi-task iterative tool honing

SMITH is a *multi-task* RL framework that jointly trains two complementary skills: *tool creation* (build) and *tool use*. In the build task the model synthesizes a reusable tool, expressed as both a Python function and an OpenAI-compatible JSON schema, a structured description of the function's name, parameters, and types that exposes the tool through a standard invocation interface. In the use task the model sees *only* this compact JSON schema (not the underlying code) and must invoke the tool to answer a held-out question. This schema-grounded design forces the model to produce concise, self-contained interfaces: a tool whose schema is ambiguous or incomplete will fail at use time, providing a direct training signal for schema quality.

We train the shared policy using DAPO [15], a clip-higher variant of GRPO [14] that stabilizes entropy and avoids reward collapse during on-policy rollouts. Each training step samples a batch $\mathcal{B}$ of prompts split into two equal halves, $|\mathcal{B}_{\text{build}}| = |\mathcal{B}_{\text{use}}| = B/2$. Keeping the two reward streams structurally separate while training on the same policy allows the two skills to reinforce each other.

Task formulation. For each training instance of tool generation, the policy receives $N{=}4$ question-answer pairs $\{(q_i, a_i)\}_{i=1}^{N}$ as context for *problem induction*: the model must infer a general solution strategy and express it as both a Python function $\mathcal{C}$ and an OpenAI-compatible JSON schema $\mathcal{S}$. A disjoint set of $K{=}16$ held-out questions $\mathcal{T} = \{(q_j, a_j)\}_{j=1}^{K}$ is then used to evaluate the generated tool without the model ever observing the ground truth answers at generation time. For use tasks, the model instead receives a single target question and must invoke a tool from the pool (if one exists for the category) or first build one from the same N in-context examples before invoking it.

3.1 Build task rewards

Evaluation reward. The generated tool $(\mathcal{C}, \mathcal{S})$ is evaluated against the hidden test set $\mathcal{T}$. Each question q_j is presented to an evaluator model π^{eval} that may call the generated tool; a correct answer requires both a successful tool invocation and an output verified by an LLM equivalence judge. π^{eval}

is initialized from the same base checkpoint as the policy and is periodically refreshed by copying the latest policy weights, providing a stable but improving evaluation target without the instability of evaluating against the live training weights. The evaluation reward is the fraction of test questions answered correctly:

$$r^{\text{eval}} = \frac{1}{|\mathcal{T}|} \sum_{j=1}^{|\mathcal{T}|} \mathbf{1}\big[\pi^{\text{eval}}(q_j \mid \mathcal{C}, \mathcal{S}) \approx a_j\big]. \tag{1}$$

Counting only answers obtained through a successful tool call prevents the policy from exploiting the fallback of text-only reasoning, which would bypass the tool-writing objective entirely.

Format reward. We apply a format reward $r^{\text{fmt}} \in \{0, r_f\}$ ($r_f = 0.5$) when the response contains exactly one Python block and one JSON block whose function names and parameter signatures are mutually consistent. If the generated response cannot be parsed into a valid $(\mathcal{C}, \mathcal{S})$ pair (e.g. missing code or schema), the rollout is terminated early and all reward axes are set to zero; $r^{\text{fmt}} = 0$ alone does not terminate the rollout. The build environment reward folds the format and evaluation signals together:

$$r^{\text{env}}_{\text{build}} = r^{\text{fmt}} + r^{\text{eval}}. \tag{2}$$

Judge reward. A separate LLM judge π^{judge} scores the generated tool on three axes: code correctness s_{code}, schema quality s_{schema}, and an overall quality score $s_{\text{overall}} \in [0, 1]$. A schema-code alignment check is applied after scoring: if the function signatures in $\mathcal{C}$ and $\mathcal{S}$ disagree, the score is halved; a syntax error in $\mathcal{C}$ yields a fixed negative reward to penalize broken code:

$$r^{\text{judge}} = \begin{cases} -0.5 & \text{if } \mathcal{C} \text{ contains a syntax error,} \\ 0.5 \cdot s_{\text{overall}} & \text{if schema and code signatures disagree,} \\ s_{\text{overall}} & \text{otherwise.} \end{cases} \tag{3}$$

Crucially, r^{judge} is *not* folded into $r^{\text{env}}_{\text{build}}$; it is passed to DAPO as an independent reward axis with its own coefficient, keeping the execution signal and the semantic quality signal disentangled. The build task therefore contributes two independent reward axes to DAPO:

$$\big(\, r^{\text{env}}_{\text{build}} = r^{\text{fmt}} + r^{\text{eval}}, \quad r^{\text{judge}} \,\big), \tag{4}$$

exposing format consistency, execution accuracy, and judge quality as the three reward signals that govern build-task learning, complemented by a single correctness axis from the use task. Any tool with $r^{\text{eval}} > 0$ is added to the shared Tool Pool $\mathcal{P}$ for reuse in subsequent use-task rollouts (Section 3.2).

3.2 Use task rewards

For use tasks the model executes a multi-turn dialogue with the tool for up to T=5 turns, terminating early when the model produces a final answer or fails to emit a valid tool call.

Correctness reward. Let $c \in \{0, 1\}$ indicate whether the final answer matches the ground truth a^*, verified by string normalization, with an LLM equivalence judge as a fallback. To penalize turn-budget exhaustion, the correctness score is scaled by an efficiency multiplier $\eta(\rho)$ that decays as the turn fraction $\rho = \min(n/T, 1)$ grows. We define a minimum reward floor $\eta_{\min}$ to ensure the policy is never strictly indifferent to correctness, even at the turn limit:

$$r^{\text{correct}} = 2\, c \cdot \eta(\rho), \qquad \eta(\rho) = \begin{cases} 1 - 2(1 - \eta_{\text{mid}})\rho & \rho \le 0.5, \\ \max\big(\eta_{\min},\ \eta_{\text{mid}}\,(1 - 2(\rho - 0.5))^2\big) & \rho > 0.5. \end{cases} \tag{5}$$

The piecewise form is continuous at $\rho = 0.5$ by construction: both branches evaluate to η_{mid} at the boundary. We set $\eta_{\min} = 0.3$ and $\eta_{\text{mid}} = 0.7$; with these values the floor activates at $\rho \approx 0.83$ (after 5 turns). These values were chosen based on small-scale preliminary runs and held fixed throughout all experiments. The use task therefore contributes a single reward axis to DAPO : r^{correct} the efficiency-weighted final-answer correctness defined in Eq. 5; tool-execution success is implicit, since correctness requires a valid tool invocation. Together with the two build axes (Eq. 4), this gives three independent reward axes for joint policy optimization.

Joint policy update. Each training batch is a disjoint union $\mathcal{B} = \mathcal{B}_{\text{build}} \sqcup \mathcal{B}_{\text{use}}$ with $|\mathcal{B}_{\text{build}}| = |\mathcal{B}_{\text{use}}| = B/2$, where B is the per-iteration batch size. This keeps the gradient contribution from each task balanced. DAPO computes per-prompt advantages independently within each generation group and accumulates gradients across both task types in a single backward pass:

$$\mathcal{L} = \mathcal{L}_{\text{DAPO}}(\mathcal{B}_{\text{build}}) + \mathcal{L}_{\text{DAPO}}(\mathcal{B}_{\text{use}}). \tag{6}$$

Because both losses share the same policy parameters θ, gradients from build and use prompts jointly update the model in every step, with no explicit advantage combination across task types.

4 Experiments

4.1 Training tasks

We train on **13 task categories** from Reasoning-Gym (RG) [18], selected for three properties: answers are exact and automatically verifiable (enabling reward computation without human annotation), each task exposes a curriculum of difficulty levels (enabling the easy-to-hard protocol described in Sec. 4.4), and questions can be generated procedurally. The categories span arithmetic (*bitwise arithmetic*, *cryptarithmetic*), algorithms (*bit counting*, *LCM*, *GCD*, *base conversion*, *isomorphic string*), algebra (*polynomial equations*, *polynomial multiplication*), games (*countdown*, *Tower of Hanoi*), and logical reasoning (*knights-and-knaves*, *Caesar cipher*). This design separates tool-writing quality from instance difficulty: a correct tool induced on easy examples must generalize to hard examples without further adaptation, a separation unavailable in fixed-label benchmarks where harder instances cannot be generated on demand.

4.2 Evaluation benchmarks

We measure transfer at three levels of increasing distance from training : RG (Seen) and RG (Unseen).

RG (Seen). Macro-average accuracy over the 13 training task categories, evaluated at the hardest curriculum level. This measures whether the learned policy produces tools that generalize beyond easy induction contexts to harder instances of the same task families.

RG (Unseen). Macro-average accuracy over **10 RG tasks** withheld entirely from RL training, spanning: arithmetic (*calendar arithmetic*, *complex arithmetic*, *time intervals*); algebra (*Chinese remainder theorem*, *simple equations*); algorithms (*group anagrams*); and logic & games (*ab*, *self-referential sequence*, *syllogism*, *Puzzle-24*). While these categories appear in RG (Seen), the specific problem types are entirely absent from training, probing cross-task generalization within Reasoning-Gym.

Out-of-domain benchmarks. TabMWP-Hard: a strengthened variant of TabMWP [29]; the original is trivially solved in CoT (96.8% EM) because tables average < 10 rows and 2 columns, so we extend rows to up to 5,000 and add unrelated columns (Appendix M). **GQA** [30]: visual question answering, requiring visual tools. Together these cover the three most relevant tool-use classes (computation, knowledge access, non-textual modality) per Wang et al. [31].

4.3 Models and training configuration

We use QWEN3-4B-INSTRUCT [32] as the primary target for its strong instruction-following and tool use ability. For all training, we fine-tune with LoRA (r=64, α=128) to reduce the training compute. We train with DAPO [15] for 60 gradient steps across 13 task categories, with generation and use tasks at a 1:1 ratio within each batch, n_{gen}=8 rollouts per prompt, temperature 0.7, β=0.01 (KL coefficient), learning rates 6×10^{-5} (4B) and 1×10^{-5} (8B). The Tool Pool (Appendix E) caches up to 20 verified tools per category; each use-task rollout injects 1 domain tool (from the correct category) and 2 distractor tools (from unrelated categories, forcing the model to identify and invoke the correct schema) into the prompt.

4.4 Train/test split design for easy-to-hard transfer

We separate the difficulty of tool induction from the difficulty of tool evaluation. For each RG task, the *induction context* is drawn from the easier curriculum bands (train), while the *tool evaluation set* is

drawn from the hardest band (test); the exact mapping is task-specific. The reward r^{eval} (Sec. 3.1) is therefore sharply reduced for any tool that merely pattern-matches the easy induction context without abstracting the underlying algorithm, pushing the policy toward tools that are concise, interpretable, and reusable. Unlike prior inference-time tool-generation frameworks [9–12], which evaluate tools on the same difficulty distribution used to create them, our protocol forces an easy-to-hard generalization gap at every step.

The complete per-task split for all 13 categories is in Appendix F.

4.5 Baselines

All baselines use QWEN3-4B-INSTRUCT unless noted. **Standard CoT** provides the no-tool ceiling. However, in GQA since LLMs cannot process visual information without external visual tool, the LLMs can only answer based on its knowledge learned from textual world.

LATM [9], **CRAFT** [10], **Trove** [11], and **KTCE** [12] all create tools at inference time with a frozen Qwen3-4B-Instruct backbone, differing only in scaffolding complexity; comparing them on the same backbone tests whether RL training adds value beyond prompt engineering.[3] We additionally evaluate **LATM with Qwen3-30B-A3B** as a deliberate scaling probe, isolating whether sheer model size on the same inference-time harness can close the gap to RL training. **ReTool (4B distill Qwen-32B)** [24] is a multi-turn code-execution policy (up to 10 turns) distilled from Qwen-32B traces that never produces a reusable schema; including it isolates the contribution of the schema-grounded tool representation, since both approaches use execution feedback but only SMITH produces reusable callable schemas. **LATM (4B distill GPT-4.1)** fine-tunes the backbone on tool-writing trajectories generated by GPT-4.1; comparing against it tests whether RL training over an explicit reward signal yields better tools than behavioral cloning from a stronger frozen oracle.

Evaluation protocol. We report macro-average accuracy with two Reasoning-Gym averages: *RG (Seen)* over the 13 training categories and *RG (Unseen)* over the 10 held-out categories.

4.6 Main results

We compare SMITH against inference-time tool-creation baselines (LATM, CRAFT, TroVE, KTCE), distillation baselines (ReTool 4B distilled from Qwen-32B; LATM 4B distilled from GPT-4.1), and a larger model (Qwen3-30B-A3B) on LATM, all on the same evaluation protocol. Two findings stand out in Table 1. First, against distillation: SMITH (4B, RL) generalizes more reliably than 4B models distilled from much larger oracles, attaining the highest held-out RG accuracy overall (79.9) versus 63.2 for ReTool and 65.8 for LATM-distilled. ReTool in particular leads on RG (Seen) (92.2 vs. our 85.2) but loses nearly 30 points on RG (Unseen), indicating that rejection-sampled distillation overfits to the demonstrator's training distribution rather than learning a transferable build-and-use policy. Second, against more elaborate scaffolds: SMITH's simple scaffold LATM is simply write multiple versions of tools and then pick the best, is the strongest results on RG (Seen) at 85.2 and beats every scaffolding baseline on RG (Unseen), including CRAFT (76.5), KTCE (65.1), and TroVE (55.9). In addition to beating larger models Qwen3-30B-A3B Instruct and a distilled version of Qwen3 4B from GPT-4.1 responses through rejection sampling finetuning. These results suggest that learning *how* to build a tool from a verifiable reward outperforms both behavioral cloning from a stronger oracle and hand-engineered retrieval/refinement loops.

Third, on token efficiency: SMITH achieves the strongest aggregate accuracy with the smallest output budget at 100 tokens on average, roughly $32\times$ fewer than Standard CoT (3,206) and $6\times$ fewer than ReTool (633), while input usage remains modest at 664 tokens, comparable to LATM (607) and well below CRAFT (1,226) and ReTool (1,707). The asymmetry is informative: scaffolding baselines such as CRAFT and the distillation-trained ReTool spend far more input tokens on retrieved exemplars or stitched prompts, and CoT spends its budget on long unconditioned reasoning, yet none recovers SMITH's holdout accuracy. This shows that the RL objective shifts the work out of decode-time reasoning and into reusable tool code, so each query is resolved by a short tool invocation rather than a long chain of thought.

[3] We rewrite the LATM prompt for better schema clarity.

Table 1: Reasoning-Gym results (Qwen3-4B-Instruct); CRAFT, TroVE, KTCE, ReTool, LATM-distill, and SMITH report mean±std over seeded re-evaluations. SMITH leads held-out generalization (RG Unseen 79.9) over distilled models (ReTool 63.2, LATM-distill 65.8) and every scaffolding baseline, using 32× fewer output tokens than standard CoT. Fixing baseline grading bugs raises TroVE's RG (Unseen) from 40.6 to 55.9 and KTCE's from 48.7 to 65.1. *RG (Seen/Unseen)*: macro-average over training/held-out tasks. *LATM prompt rewritten for schema clarity. I/O: input / output tokens.

	Seen	Unseen RG						
Method	**Avg**	**Logic**	**Game**	**Algebra**	**Arith**	**Algo**	**Avg**	**I/O**
Standard CoT	58.0	49.9	60.3	56.8	62.7	48.6	55.7	173 / 3,206
LATM* [9]	77.6	53.9	55.5	38.6	53.0	90.2	58.3	607 / 174
LATM* - Qwen3-30B-A3B	74.0	68.7	64.2	97.3	56.5	84.0	74.1	659 / 405
CRAFT [10]	74.1±0.7	27.4±1.3	**89.5±0.0**	94.2±1.2	76.6±1.1	95.0±0.0	76.5±0.4	1,226 / 418
Trove [11]	52.6±0.4	60.7±2.4	10.6±1.3	51.2±3.5	59.8±0.8	**97.0±0.0**	55.9±0.6	347 / 575
KTCE [12]	61.0±1.5	60.2±1.5	79.8±1.6	70.6±0.3	45.6±0.4	69.3±1.8	65.1±0.2	319 / 404
ReTool (distill Qwen-32B)	**92.2±0.8**	50.3±2.3	55.0±0.8	48.7±0.6	**79.8±2.7**	82.4±0.1	63.2±0.4	1,707 / 633
LATM (distill GPT-4.1)	81.7±4.4	37.6±15.2	58.1±12.2	91.3±7.7	51.6±10.4	93.2±5.9	65.8±4.1	638 / 207
SMITH	85.2±2.7	**74.2±0.6**	63.7±1.1	**97.9±2.6**	70.6±2.1	93.0±0.4	**79.9±2.2**	664 / 100

Table 2: Tools from SMITH's RL-trained 4B writer enable a 350M model (LFM2.5) to match a 30B tool writer on held-out tasks (42.9 vs. 41.5 RG Unseen), showing that SMITH's tools encode genuinely generalizable solutions that transfer across model families. First row is the no-tool baseline. RG (Seen)/(Unseen): macro-average accuracy (%). Best per column in **bold**.

Method	**RG (Seen)**	**RG (Unseen)**	**TabMWP**	**GQA**
LFM2.5-350M (no tool)	14.2	11.6	0.0	0.1
+ Qwen3-4B-Instruct (tool writer)	36.8	23.4	4.2	0.1
+ Qwen3-30B-A3B (tool writer)	10.4	41.5	4.1	0.1
+ RL on LFM2.5-350M	**39.1**	30.2	4.2	0.1
+ RL 4B (Ours)	38.9	**42.9**	4.1	0.1

For all baselines TroVE, CRAFT and KTCE we did manually inspect the original codebase and found some python code parsing issues when running on Qwen3-4B. Hence we have fixed those issues in our revision. We also tried to modify the TroVE original prompts and we found it did not perform any better or significantly worse than the original prompt.

4.7 Tools transfer across model scale

A natural question is whether the tools synthesized by SMITH encode genuinely generalized solutions or whether they only work with the same policy model that wrote them. We test both directions: pairing SMITH's 4B writer with a much smaller consumer and with a much larger one.

Smaller student. We pair our best fine-tuned 4B tool generation model with LFM2.5-350M [33], a 350 M-parameter small model with strong tool use ability, to use those tools at inference time. Table 2 shows that pairing LFM2.5-350M with our RL 4B tool-generation model increases holdout RG accuracy from 11.6 to 42.9, matching the much larger Qwen3-30B-A3B-Instruct writer (41.5); in training tasks, our 4B model (21.1) actually exceeds the 30B untrained model (10.4). We also train SMITH model using LFM2.5 as reward signal on tool generation, while it performs the best in RG Seen set but underperforms on RG Unseen.

Larger consumer. The complementary question is whether the 4B writer remains useful once a much stronger tool user is available, the practical deployment setting where one could simply let the larger model write its own tool instead. We pair tools generated by our RL-trained 4B writer with Qwen3-30B-A3B-Instruct as the tool *consumer* and compare against LATM*, where the same 30B model both writes and uses its own tool, across the 25 tasks the two configurations share (10 held-out RG, 13 seen RG, TabMWP-Hard, GQA). Table 3 shows the SMITH-4B-written tools lift the 30B consumer on every group, most sharply on TabMWP-Hard (0.7 → 38.8), and raise the task-weighted

Table 3: Tools written by SMITH's RL-trained 4B writer lift a Qwen3-30B-A3B-Instruct consumer above the same 30B model writing tools for itself (LATM*) on every group, with the largest gain on TabMWP-Hard (0.7 → 38.8), showing the 4B writer is a viable drop-in tool provider even when a much larger model is available to consume its tools. *Overall*: task-count-weighted average over the 25 shared tasks (10 held-out RG + 13 seen RG + TabMWP-Hard + GQA). Best per column in **bold**.

Method	Held-out RG (10 tasks)	Seen RG (13 tasks)	TabMWP-Hard	GQA	Overall
LATM* (30B writes its own tool)	71.9	78.1	0.7	20.2	70.2
SMITH-4B to 30B consumer	**74.5**	**84.6**	**38.8**	**30.2**	**76.6**

Table 4: OOD generalization (Qwen3-4B-Instruct). SMITH leads on TabMWP-Hard (40.4 vs. next best TroVE 36.4) and ranks second on GQA (42.6), making it the only 4B method without oracle distillation to top either column. Bold = best, underline = second-best.

	Baselines						Distilled		
Task	CoT	LATM	LATM* (30B)	CRAFT	TroVE	KTCE	ReTool	LATM (distill)	SMITH
TabMWP-Hard	7.2	19.7	0.7	30.0	<u>36.4</u>	27.2	3.0	7.1	**40.4**
GQA	11.5	35.0	29.8	21.9	21.4	0.0	26.1	**56.0**	<u>42.6</u>

overall score from 70.2 to 76.6. A stronger tool user therefore does not make its own self-written tool preferable: the RL-trained 4B writer's tools remain a better source of tools than the 30B model's own. Taken together, these two results show that SMITH's tools transfer in both directions, down to a 350M model and up to a 30B model, so the 4B writer is a viable drop-in tool provider regardless of which model ultimately consumes its tools.

4.8 Out-of-distribution evaluation

To probe transfer beyond Reasoning-Gym, we evaluate on a tabular-reasoning benchmark (TabMWP-Hard) and a visual-reasoning benchmark (GQA), neither of which is represented during training. Table 4 shows SMITH leading TabMWP-Hard at 40.4, ahead of the closest baseline TroVE (36.4), and reaching second on GQA at 42.6, trailing only LATM distilled from GPT-4.1 (56.0). We do not claim parity on perception: GPT-4.1 distillation embeds visual primitives our self-trained writer never sees during RL, and the gap is the cost of avoiding oracle supervision. What *does* transfer without distillation is the self-supervised tool-creation loop. SMITH outperforms every scaffolding baseline on both OOD benchmarks and is the only 4B same-backbone method without oracle distillation to lead either column.

4.9 Scaling across backbones

To test whether SMITH's training signal transfers across model sizes and families, we apply the same RL recipe to QWEN3-8B and GRANITE-3.3-8B. Table 5 shows that for Qwen3-8B, SMITH improves both the accuracy in-distribution (72.6 → 79.4) and the holdout RG accuracy (72.2 → 81.7) and lifts the OOD TabMWP-Hard from 42.4 to 56.7. While Granite-3.3-8B starts from a much weaker base (RG Seen 31.2, Unseen 22.0) yet follows the same trend: SMITH lifts RG (Seen) to 39.1, RG (Unseen) to 28.5, and GQA from 7.8 to 11.7.

We further test whether SMITH still works when the judge signal comes from the policy itself rather than an external 30B-A3B model. The **Self-Judge** variant prompts Qwen3-8B with the same judge template used for 30B-A3B, scoring its own rollouts. Self-judging improves held-out RG (81.7 → 85.9, the best in the table) but trades off in-distribution accuracy and OOD GQA (28.7 → 16.3), suggesting the smaller judge is a weaker but less biased signal on training-task distributions.

Table 5: SMITH improves over the base model on every backbone tested. On Qwen3-8B it raises held-out RG from 72.2 to 81.7 and TabMWP-Hard from 42.4 to 56.7; Granite-3.3-8B follows the same trend despite a weaker starting point, confirming the training signal is not model-family specific. *RG (Seen/Unseen)*: macro-average over training/held-out tasks. Best per column in **bold**.

Method	RG (Seen)	RG (Unseen)	TabMWP	GQA
QWEN3-8B (baseline)	72.6	72.2	42.4	17.3
SMITH: Qwen3-8B	**79.4**	**81.7**	**56.7**	**28.7**
SMITH: Self-Judge	**74.7**	**85.9**	54.5	16.3
GRANITE-3.3-8B (baseline)	31.2	22.0	3.9	7.8
SMITH: Granite-3.3-8B	39.1	28.5	4.5	11.7

Table 6: SMITH improves external tool-calling (BFCL v4, no-web) across all backbones, with the largest gain on Qwen3-8B (43.3→55.8), despite never seeing BFCL schemas, multi-turn traces, or judges during RL training. Bold = best, underline = second-best.

	Qwen3-4B-Instruct		Qwen3-8B		Granite-3.3-8B	
Metric	**Base**	**SMITH**	**Base**	**SMITH**	**Base**	**SMITH**
BFCLv4	45.1	**48.6**	43.3	**55.8**	36.3	**38.7**

4.10 Generalization to external tool-calling

Beyond benchmarks where the model writes its own tools, we test whether SMITH's build-and-use objective transfers to *externally specified* function-calling APIs using BFCL v4 (no-web subset) [34]. Table 6 shows that SMITH lifts BFCL overall accuracy on both Qwen backbones, from 45.1 to 48.6 on Qwen3-4B and from 43.3 to 55.8 on Qwen3-8B, the largest absolute gain in the table. Since BFCL schemas, multi-turn traces, and judges are never seen during RL, the improvement isolates a learned tool-use prior rather than benchmark-specific fitting.

4.11 Ablation

To isolate the factors driving SMITH's gains, we ablate the reward structure on Qwen3-4B-Instruct and compare against four configurations: (1) TOOL CREATE, where the reward depends only on the quality of the generated code; (2) DECOUPLED BUILD/USE, which pairs the row-1 30B-A3B builder with a separately trained 4B tool-use specialist, testing whether joint training (not specialization) is the active ingredient; (3) SMITH: NO LLM JUDGE, which couples tool creation with execution success but drops the judge signal; and (4) the full SMITH objective, which keeps the LLM-as-judge signal as a separate axis. Table 7 shows that each component contributes incrementally: tool creation alone yields a strong RG (Seen) bump but leaves OOD GQA underperforming; the decoupled pair fails to beat single-model TOOL CREATE on RG Unseen (58.9 vs. 68.8), confirming that joint training, not specialization, drives the gain; coupling build and use jointly lifts in-distribution accuracy but hurts held-out transfer; only the full SMITH objective achieves the best aggregate score, indicating that disentangling process quality from outcome correctness is essential for cross-domain robustness.

5 Limitations and discussion

Scale, judge dependence, and base priors. All trained policies are at most 8B parameters and the quality judge has 30B activated parameters, a regime chosen for compute feasibility; whether the gains persist, saturate, or invert at the 70B+ scale is an open question, and our Self-Judge ablation (Table 5) is only a partial probe of judge-size dependence, since dropping the external judge improves held-out RG but degrades OOD GQA. Finally, SMITH executes generated Python at every step inside a sandbox, but we do not formally certify that adversarial prompts cannot induce unsafe tools, and we report no human evaluation of schema readability or developer-facing reusability.

Open Questions. 1. In SMITH, a tool is one Python function plus a JSON schema. Richer artifacts are not evaluated in this work. For example, skills package workflow instructions in a `SKILL.md` file

Table 7: Reward-structure ablation (Qwen3-4B-Instruct). π^{eval}: model that invokes the generated tool at eval (*30B-A3B*/*4B*: separate model; *self*: same RL'd policy). **Tool Use**: whether π^{eval} calls the tool rather than only writing it. Decoupled build/use does not beat single-model TOOL CREATE on RG (Unseen) (58.9 vs. 68.8), so joint training, not specialization, drives the gain; only the full SMITH objective wins on *aggregate*, with individual benchmarks split between K=1 (TabMWP) and NO LLM JUDGE (GQA). Best per column **bold**, second-best underlined. RG(S)/(U): Seen/Unseen.

Method	π^{eval}	Tool Use	RG (S)	RG (U)	TabMWP	GQA
QWEN3-4B-INSTRUCT	–	–	61.85	47.01	19.70	20.86
TOOL CREATE	30B-A3B	No	77.43	59.39	15.60	37.01
TOOL CREATE	4B	No	73.88	68.80	17.70	35.82
DECOUPLED CREATE/USE	30B-A3B	Yes	76.44	58.93	13.90	35.04
SMITH : NO SYNC	4B	Yes	80.29	66.89	25.81	24.58
SMITH : NO LLM JUDGE	self	Yes	82.57	67.81	18.30	**42.63**
SMITH : K=1	self	Yes	78.64	73.93	**46.77**	32.30
SMITH : FULL	self	Yes	**86.61**	**78.33**	40.40	42.62

together with optional scripts, references, templates, and other resources. MCP servers, by contrast, use the Model Context Protocol to expose tools, resources, prompts, and instructions to a model; they are not themselves bundles of procedures and scripts. Supporting the generation of such multi-file or server-backed artifacts would require extending SMITH's output representation and validation pipeline. Iterative or multi-turn generation may be useful for this setting, but is not inherently required. We leave this extension to future work.

2. Throughout SMITH training, we never observed the model issue parallel tool calls in a single turn or generate multiple tools at once. We suspect this reflects both the base model's tendency toward single-tool use and our reward design, which does not encourage multi-tool generation or parallel execution. Since parallel calls are common in real-world settings such as deep research, we believe SMITH could naturally extend to these settings, which we leave to future work.

6 Conclusion

We introduced SMITH, a reinforcement learning framework that jointly trains a single language model to *create* and *use* reusable tools, closing the feedback loop between tool writer and tool user so the policy is optimized directly on its own execution outcomes. Trained on 13 Reasoning-Gym tasks, our 4B model attains the highest held-out RG accuracy among all evaluated methods (79.8), leads TabMWP-Hard, and writes tools that transfer to a 350 M student never seen during training, matching the quality of tools produced by a model an order of magnitude larger. The same recipe lifts Qwen3-8B and Granite-3.3-8B without modification, indicating that coupling creation and use inside a single trained policy is a scalable path to generalization: the tools a model writes become precisely the tools it can reliably invoke, without a larger frozen teacher, a more complex scaffold, or out-of-domain supervision.

Acknowledgments

This work was supported in part by the National Science and Technology Council, Taiwan, under the Grants 115-2628-E-002-023-MY4, 112-2223-E-002-012-MY5, 115-2628-E-002-006, 115-2223-E-002-005-MY3, and 115-2634-F-002-012, the Taiwan Centers of Excellence in Artificial Intelligence, and the Center of Data Intelligence: Technologies, Applications, and Systems, NTU (grant nos. 115L900901). Shao-Hua Sun was supported by the Yushan Fellow Program of the Ministry of Education, Taiwan.

References

[1] Edwin Hutchins. *Cognition in the Wild*. MIT Press, 1995.

[2] Andy Clark and David Chalmers. The extended mind. *Analysis*, 1998.

[3] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[4] Jeffrey Cheng, Marc Marone, Orion Weller, Dawn Lawrie, Daniel Khashabi, and Benjamin Van Durme. Dated data: Tracing knowledge cutoffs in large language models. In *First Conference on Language Modeling*, 2024.

[5] Zhibin Gou, Zhihong Shao, Yeyun Gong, Yujiu Yang, Minlie Huang, Nan Duan, Weizhu Chen, et al. Tora: A tool-integrated reasoning agent for mathematical problem solving. In *International Conference on Learning Representations*, 2024.

[6] Mojtaba Komeili, Kurt Shuster, and Jason Weston. Internet-augmented dialogue generation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, 2022.

[7] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*, 2022.

[8] Ross Taylor, Marcin Kardas, Guillem Cucurull, Thomas Scialom, Anthony Hartshorn, Elvis Saravia, Andrew Poulton, Viktor Kerkez, and Robert Stojnic. Galactica: A large language model for science. *arXiv preprint arXiv:2211.09085*, 2022.

[9] Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In *International Conference on Learning Representations*, 2024.

[10] Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi Fung, Hao Peng, and Heng Ji. Craft: Customizing llms by creating and retrieving from specialized toolsets. In *International Conference on Learning Representations*, 2024.

[11] Zhiruo Wang, Graham Neubig, and Daniel Fried. Trove: Inducing verifiable and efficient toolboxes for solving programmatic tasks. In *International Conference on Machine Learning*, 2024.

[12] Zhiyuan Ma, Zhenya Huang, Jiayu Liu, Minmao Wang, Hongke Zhao, and Xin Li. Automated creation of reusable and diverse toolsets for enhancing llm reasoning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2025.

[13] Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*, 2022.

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

[15] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. In *Advances in Neural Information Processing Systems*, 2025.

[16] Arjun Panickssery, Samuel R. Bowman, and Shi Feng. LLM evaluators recognize and favor their own generations. In *Advances in Neural Information Processing Systems*, 2024.

[17] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-bench and chatbot arena. In *Advances in Neural Information Processing Systems*, 2023.

[18] Zafir Stojanovski, Oliver Stanley, Joe Sharratt, Richard Jones, Abdulhakeem Adefioye, Jean Kaddour, and Andreas Köpf. Reasoning gym: Reasoning environments for reinforcement learning with verifiable rewards. In *Advances in Neural Information Processing Systems*, 2025.

[19] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, 2023.

[20] Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *Transactions on Machine Learning Research*, 2023.

[21] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, 2022.

[22] Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. Rlef: Grounding code llms in execution feedback with reinforcement learning. In *International Conference on Machine Learning*, 2025.

[23] Cheng Qian, Emre Can Acikgoz, Qi He, Hongru WANG, Xiusi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. Toolrl: Reward is all tool learning needs. In *Neural Information Processing Systems*, 2025.

[24] Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. Retool: Reinforcement learning for strategic tool use in llms. *arXiv preprint arXiv:2504.11536*, 2025.

[25] Jiongxiao Wang, Qiaojing Yan, Yawei Wang, Yijun Tian, Soumya Smruti Mishra, Zhichao Xu, Megha Gandhi, Panpan Xu, and Lin Lee Cheong. Reinforcement learning for self-improving agent with skill library. *arXiv preprint arXiv:2512.17102*, 2025.

[26] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, 2023.

[27] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. In *International Conference on Machine Learning*, 2024.

[28] Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. Inducing programmatic skills for agentic tasks. In *Second Conference on Language Modeling*, 2025.

[29] Pan Lu, Liang Qiu, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, Tanmay Rajpurohit, Peter Clark, and Ashwin Kalyan. Dynamic prompt learning via policy gradient for semi-structured mathematical reasoning. In *International Conference on Learning Representations*, 2023.

[30] Drew A Hudson and Christopher D Manning. Gqa: A new dataset for real-world visual reasoning and compositional question answering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019.

[31] Zhiruo Wang, Zhoujun Cheng, Hao Zhu, Daniel Fried, and Graham Neubig. What are tools anyway? a survey from the language model perspective. In *First Conference on Language Modeling*, 2024.

[32] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[33] Liquid AI. Lfm2 technical report. *arXiv preprint arXiv:2511.23404*, 2025.

[34] Shishir G. Patil, Huanzhi Mao, Charlie Cheng-Jie Ji, Fanjia Yan, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *International Conference on Machine Learning*, 2025.

[35] Matthias Minderer, Alexey Gritsenko, Austin Stone, Maxim Neumann, Dirk Weissenborn, Alexey Dosovitskiy, Aravindh Mahendran, Anurag Arnab, Mostafa Dehghani, Zhuoran Shen, et al. Simple open-vocabulary object detection. In *European conference on computer vision*, pages 728–755. Springer, 2022.

[36] Junnan Li, Dongxu Li, Caiming Xiong, and Steven Hoi. Blip: Bootstrapping language-image pre-training for unified vision-language understanding and generation. In *International conference on machine learning*, pages 12888–12900. PMLR, 2022.

Appendix

Table of Contents

A Broader Impact **15**

B Lessons Learned in Designing Verifier Rewards and LLM-as-Judge **15**
B.1 System Overview 15
B.2 Observed Failure: `build/env_reward` Collapse 16
B.3 Root Cause Analysis 16
B.4 Lessons for Reward and Judge Design 17
B.5 Summary of Design Principles 20

C LLM-as-Judge Prompt (v5.0) **20**

D Build and Use Task Prompt Templates **25**
D.1 Build-Task Prompt 25
D.2 Use-Task Prompt 25
D.3 Evaluator Prompt for r^{eval} 26

E Tool Pool Design **26**
E.1 Pool Structure 26
E.2 Admission Criterion 26
E.3 Capacity Cap and Eviction Policy 27
E.4 Retrieval and Distractor Selection 27
E.5 Initialisation 27

F Full Easy-to-Hard Difficulty Split for All 13 Training Tasks **27**

G OOD Inference Protocol for TabMWP-Hard and GQA **29**
G.1 Single Build-Then-Use Loop 29
G.2 In-Context Examples 29
G.3 Protocol Differences from the RG Training Loop 29

H GQA Visual Tool Setup **30**
H.1 Visual Primitive Functions 30
H.2 Image Encoding 30
H.3 Build-Task Prompt for GQA 30
H.4 Tool Selection and Evaluation 31
H.5 Answer Verification 31

I ReTool Evaluation Setup **31**
I.1 Checkpoint and Training Configuration 31
I.2 Evaluation Protocol on Reasoning-Gym and OOD Benchmarks 32
I.3 Key Distinction from SMITH 32

J LATM (distill GPT-4.1) Baseline Setup **32**
J.1 Trajectory Collection 32
J.2 Fine-Tuning Configuration 33
J.3 Evaluation Protocol 33
J.4 Comparison to SMITH 33

K Agentic Loop Comparison: KTCE, CRAFT, TroVE, and SMITH **34**
K.1 Positioning Overview 34
K.2 KTCE: Offline Evolutionary Creation with Online Retrieval-then-Solve 34
K.3 CRAFT: Offline Diversity-Sampled Creation with Multi-View Retrieval 35
K.4 TroVE: Online Streaming Creation with Frequency-Based Forgetting 35

K.5 SMITH: RL-Trained Tool Creation Coupled with Tool Use 35
K.6 Full Comparison Table 36
K.7 Key Conceptual Distinctions 37

L Baseline Failure Analysis **37**
L.1 KTCE: Three Compounding Failure Modes 38
L.2 TroVE: Missing Token-Usage Records 38

M TabMWP-Hard: Dataset Construction and Augmentation **38**
M.1 Table Type Coverage 38
M.2 Augmentation Pipeline 39
M.3 Worked Example A: TWO-WAY Table 39
M.4 Worked Example B: PRICE LIST Table (Missing Header) 40
M.5 LLM Column Proposal Mechanism 41
M.6 Collision Avoidance 41

N Compute and Training Cost **41**

A Broader Impact

SMITH trains a single model to both author and invoke tools, which lowers the engineering effort required to extend an LLM agent into new domains: practitioners no longer need to hand-write API schemas and reference implementations for every task. We see two main positive impacts. First, smaller open-weight models become more competitive at agentic reasoning, which reduces dependence on closed frontier APIs and the associated cost and privacy trade-offs. Second, jointly training the writer and the user surfaces schema and code failures during training rather than at deployment, producing tools that are more reliable for downstream users than those produced by inference-time prompting pipelines. On the risk side, an LLM that autonomously synthesises and executes code raises the usual sandboxing concerns: a model that writes its own tools can also write tools that are unsafe to run on a host with broad filesystem or network access. Our training and evaluation environments execute generated code in restricted Python sandboxes with no network access and a limited standard library, and we recommend the same posture for any deployment built on this recipe. We do not see additional dual-use risk beyond what is already present in code-generating LLMs: the tools learned in this work target procedural reasoning (arithmetic, calendars, table lookups, visual question answering), not capabilities that materially uplift cyber, biological, or other restricted domains.

B Lessons Learned in Designing Verifier Rewards and LLM-as-Judge

B.1 System Overview

The training system fine-tunes Qwen3-4B-Instruct on two interleaved task types with DAPO loss and LoRA ($r = 64$). In **build tasks**, the model writes a Python function and a matching OpenAI-compatible JSON schema, validated and scored by three independent signals: a structural verifier, a LoRA-synced evaluator that runs held-out test questions through the generated tool, and an LLM judge scoring code quality. In **use tasks**, the model is given a pre-built tool schema and must invoke it correctly to answer a question; reward is rule-based answer matching.

In the configuration analysed here (which differs from the final system), the LLM judge was a *fixed* external model (Qwen3-30B) rather than the self-synced policy checkpoint used in the final SMITH design.

The total reward for a build trajectory is:

$$r_{\text{build}} = \underbrace{r_{\text{format}}}_{\text{verifier}} + \underbrace{r_{\text{eval}}}_{\text{LoRA evaluator}} + \underbrace{w_j \cdot r_{\text{judge}}}_{\text{LLM judge}}, \quad w_j = 0.5 \tag{7}$$

where $r_{\text{format}} \in \{0, 0.5, 1.0\}$ encodes schema–function-name alignment (0.5) and parameter alignment (0.5), $r_{\text{eval}} \in [0, 1]$ is the fraction of test questions answered correctly by the generated tool, and $r_{\text{judge}} \in [-0.5, 1]$ is the normalised LLM judge score. Note that the final system tightens r_{format} to a binary $\{0, 0.5\}$ signal granted only when both function-name and parameter alignment hold simultaneously (Section 3.1); the additive $\{0, 0.5, 1.0\}$ form above is specific to the configuration analysed in this appendix.

B.2 Observed Failure: `build/env_reward` Collapse

Quantitative trajectory. Table 8 shows per-step metrics logged during training. The LoRA sync occurs at step 5 (checkpoints are saved every 5 steps). A sharp regression is visible from step 6 onward.

Table 8: Key build-task metrics per optimizer step. The LoRA sync that loads `eval_lora_step5` into the evaluator server fires *after* step 5's metrics are logged. Steps 6–7 are the first rollouts with the synced evaluator.

Step	env_reward	eval_reward	step1_fail	fn_mismatch	param_mismatch	judge_reward
1	1.363	0.386	34.9%	23.4%	11.5%	0.460
2	1.809	0.466	10.4%	8.3%	2.1%	0.614
3	2.026	0.557	2.1%	1.0%	1.0%	0.670
4	2.043	0.684	9.4%	5.2%	3.6%	0.710
5	2.123	0.662	2.6%	0.0%	2.6%	0.722
LoRA sync applied after step 5						
6	1.641	0.500	23.9%	13.5%	10.4%	0.561
7	1.402	0.332	28.6%	14.1%	14.1%	0.453

Attribution of the drop. Since r_{eval} is averaged over all build states including step1 failures (which receive $r_{\text{eval}} = 0$), the r_{eval} collapse follows directly from the `step1_fail` spike. We verified this by inspecting rollout event logs: at step 6, 46 out of 238 build states failed step1 (19.3%), versus 5 out of 197 at step 5 (2.5%). Of those 46 failures, 26 were `schema_function_mismatch` and 20 were `schema_param_mismatch`.

Crucially, the *evaluator was not failing*. No connection errors or exceptions appeared during tool evaluation. Items that did reach the evaluator scored 5.26/8 test questions on average at step 6, close to the 5.44/8 at step 5. The drop in `build_eval_reward` is almost entirely explained by the increased fraction of step1-failed states receiving a forced score of zero.

B.3 Root Cause Analysis

B.3.1 Naming Drift

At step 6, the model began generating code with several plausible function names, while the schema referenced one that was not present as a top-level callable. We term this *naming drift*. Figure 2 shows a representative failure.

B.3.2 Reward Signal Misalignment

The drift persisted because the verifier and the judge applied inconsistent penalties for the same error:

Table 9: Penalty for a schema that names a function absent from the code.

Component	Condition	Penalty
Structural verifier	fn name absent	$r = 0$, trajectory terminates
LLM judge	fn name absent	$r \times 0.5$ (partial credit)

Because step1-failed trajectories are terminated before the judge scoring phase, the judge never observes naming-drift failures directly. It therefore systematically rewarded the multi-function code

```
def _gcd(a: int, b: int) -> int:
    while b:
        a, b = b, a % b
    return a

def calculate_lcm(a: int, b: int) -> int:
    return (a * b) // _gcd(a, b)

def find_lcm_of_numbers(a: int, b: int) -> int:
    return calculate_lcm(a, b)
```

```
[{"type": "function", "function": {
  "name": "find_lcm",
  "parameters": {"type": "object",
    "properties": {"a": {"type": "integer"},
                   "b": {"type": "integer"}},
    "required": ["a", "b"]}}}]
```

Figure 2: Naming drift example. The schema references `find_lcm`, which does not appear as a top-level callable in the generated code.

style that accompanied high-scoring outputs at step 5, reinforcing it via the step 5 gradient update. By step 6, the drift had progressed far enough that the schema function name was no longer present in the generated code.

Observation. *When a verifier and an LLM judge jointly determine reward, any property treated as a* hard failure *by the verifier must also yield* zero *in the judge. Partial credit for conditions that are hard failures in the verifier creates a gradient toward outputs that pass the judge but fail the verifier.*

B.3.3 Secondary Factor: LoRA-Synced Evaluator

The LoRA sync at step 5 replaces the base-model evaluator with the current training checkpoint, on the rationale that improved tool-building ability yields a stronger usability signal for r_{eval}. In practice the synced evaluator was not the primary cause of the collapse: the mean per-item test score fell only mildly (5.44 $\rightarrow$ 5.26 out of 8), while the step1-failure rate increase was the dominant driver. A brief accuracy drop immediately post-sync is nonetheless expected, as tool-use ability tends to lag behind tool-building improvements in the early steps of RL training.

Observation. *LoRA-syncing the evaluator with the training checkpoint is sound in principle, but evaluator signal quality should be monitored at each sync to distinguish evaluator degradation from policy regression.*

B.4 Lessons for Reward and Judge Design

We derive five concrete lessons from this failure.

B.4.1 Lesson 1: Hard Failures Must Be Hard Everywhere

Pitfall. *A verifier condition that zeroes out the entire trajectory reward should also zero out the judge component, not merely halve it.*

In our case, the judge applied:

The fix distinguishes the severity of each misalignment type:

Lesson. *For every binary constraint that the verifier enforces with reward $= 0$, audit the judge's scoring path and ensure it applies the same zero (or negative) for states where that constraint is violated. Partial credit for fatal structural errors creates a false gradient.*

```
if not aligned:
    if reason.startswith("syntax error in code:"):
        score = -0.5        # hard negative
    else:
        score *= 0.5        # soft halving -- fn-absent treated same
                            # as minor param mismatch
```

Figure 3: Original (buggy) misalignment handling: function-absent receives only a soft penalty.

```
if not aligned:
    if reason.startswith("syntax error in code:"):
        score = -0.5    # unparseable -- keep strong negative
    elif "not found in code" in reason:
        score = 0.0     # schema fn absent: hard zero, same as verifier
    else:
        score *= 0.5    # softer issues: param mismatch, over-promise
```

Figure 4: Fixed misalignment handling: function-absent is now a hard zero.

B.4.2 Lesson 2: Align the Judge Prompt to the Verifier's Hard Constraints

The judge prompt rewarded "descriptive naming" and "well-decomposed functions" without specifying that the schema's `name` field must exist as a top-level callable. As a result, the judge gave high `code_clarity` scores to multi-function outputs regardless of whether the schema function was present.

We extended the prompt with:

- An explicit Step 3 instruction: *"The schema `name` must appear verbatim as a top-level `def` before any other analysis. If not, assign `schema_code_alignment=0` and `overall_quality=0` immediately."*
- Two scored examples (D and E) illustrating the naming-drift anti-pattern and the thin-wrapper anti-pattern respectively.
- Updated `code_clarity` rubric language that explicitly penalises thin wrappers (score 1–2) and multi-function code where the schema-named function is not the direct implementation.

Lesson. *Every structural constraint enforced by the verifier should appear explicitly in the judge prompt, ideally with a scored counterexample. The judge cannot penalise a failure mode it has not been told to look for.*

B.4.3 Lesson 3: The Reward Gap Between Verifier and Judge Is a Gradient Leak

Let $\mathcal{S}_{\text{bad}}$ be the set of outputs that fail the verifier's hard constraint (schema function absent from code). For a GRPO update, outputs in $\mathcal{S}_{\text{bad}}$ received:

$$r_{\text{total}}(\mathbf{y}) = 0 + 0 + w_j \cdot r_j(\mathbf{y}) \quad \mathbf{y} \in \mathcal{S}_{\text{bad}}, \tag{8}$$

where the verifier and evaluator terms are both zero, but the judge term $w_j \cdot r_j > 0$ if the judge gave partial credit. If the judge assigns $r_j = 0.4$ and $w_j = 0.5$, the output in $\mathcal{S}_{\text{bad}}$ earns a total reward of 0.2 (positive), even though the verifier called it completely broken.

Under GRPO, the advantage $A(\mathbf{y}) = r_{\text{total}}(\mathbf{y}) - \bar{r}$ is positive whenever $r_j(\mathbf{y}) > 2\bar{r}$ (since $w_j = 0.5$). This means the policy is actively *reinforced* toward producing outputs in $\mathcal{S}_{\text{bad}}$ when they score well on the judge. We call this the *reward gap*: the judge's non-zero floor creates a gradient leak that partly cancels the verifier's hard zero.

Pitfall. *With N reward components combined additively, any component that assigns positive reward to verifier-failed outputs will create a gradient leak. The larger the judge weight w_j and the higher the judge's partial score for broken outputs, the stronger the gradient toward structural failures.*

Lesson. *When combining a hard verifier with a soft LLM judge, consider gating the judge signal: apply $r_j = 0$ whenever the verifier assigns $r = 0$, regardless of what the judge scores. Alternatively, use the judge signal only as a* tiebreaker *among verifier-passing outputs, not as an independent additive axis.*

B.4.4 Lesson 4: Log Structural Sub-Metrics as First-Class Signals

The aggregate reward trajectory at steps 1–5 looked healthy: `build/env_reward` was rising and `judge_reward` was stable (Table 8). The impending failure was *invisible in these aggregates*. It was only visible in the two sub-metric columns: `fn_mismatch` had been declining from 23.4% (step 1) to 0.0% (step 5), while `param_mismatch` had converged to 1–3%. A monitoring system that tracked only aggregate reward would have declared the run healthy at step 5; the structural failure at step 6 would have appeared sudden and unexplained.

Pitfall. *Aggregate reward metrics (`env_reward`, `eval_reward`, `judge_reward`) reflect the* mean *over all constraint dimensions simultaneously. A model that learns to satisfy one constraint better while silently degrading another can maintain or even improve its aggregate score until the neglected constraint crosses a critical threshold. The aggregate cannot distinguish this pattern from genuine all-round improvement.*

In our case, the correct monitoring surface was:

- `step1_fail` **rate.** The fraction of build rollouts that fail the structural verifier at the first check. This is the earliest observable symptom of naming drift or format breakdown; it should be logged every step, not inferred from `eval_reward`.
- `fn_mismatch` **rate.** The fraction of rollouts where the schema `name` field does not appear as a top-level callable in the generated code. This constraint is either satisfied or violated; its rate should stay near zero after training stabilises.
- `param_mismatch` **rate.** The fraction of rollouts where the schema's parameter list does not match the function signature. A separate, independently trackable constraint.

These sub-metrics are the *concrete example* of Principle 3 in Table 10: they make it possible to detect which specific structural constraint is degrading, and to respond with a targeted fix (as in Lesson 1 and Lesson 2) rather than a global hyper-parameter change.

A practical implementation note: because step1-failed rollouts are terminated before the evaluator and judge run, `step1_fail` is the only signal that captures them. Setting a monitoring alert on `step1_fail` $> 10\%$ would have triggered at step 6 immediately, before `eval_reward` had time to collapse all the way to the values observed at step 7.

Lesson. *Log each structural constraint-violation rate as an independent time-series metric, not as a component folded into aggregate reward. Set threshold-based alerts on violation rates, not on aggregate reward alone. When a violation rate that was decreasing suddenly spikes (even by a few percentage points), investigate immediately: this is an early warning sign of the reward-misalignment failure described in Lessons 1–3.*

B.4.5 Lesson 5: Emergent Style Transfer Can Corrupt Structural Constraints

A subtle consequence of RL with a combined reward is that the policy can learn a *style* associated with high-reward outputs and transfer that style even to outputs where it causes structural failures.

In our case, multi-function code with helper functions was correlated with high r_{eval} at steps 3–5 (better-structured tools passed more test questions). The policy learned "helper functions $\rightarrow$ high reward" as a latent heuristic. At step 6, this heuristic overrode the schema-naming constraint: the model wrote elaborate helper structures but the schema function name no longer anchored to any of them.

This is a form of *reward hacking* that is difficult to detect from the reward trajectory alone, because the style that causes the failure was associated with *correct* behaviour earlier in training. The failure is only visible in the structural sub-metrics (`fn_mismatch` rate).

Lesson. *Monitor sub-metric trajectories across training steps, not just their endpoint values. When a structural constraint violation rate that was decreasing suddenly increases, suspect that the policy has learned a correlated style feature that is generalising beyond the constraint boundary. Structural constraints should be* hard-coded *into the reward (verifier) rather than expressed through soft proxy signals (LLM judge).*

B.5 Summary of Design Principles

Table 10 summarizes the five lessons as actionable design principles for practitioners building verifier + LLM-as-judge reward pipelines.

Table 10: Design principles for combining a structural verifier with an LLM-as-judge reward signal in RL training for code generation.

#	Principle	Implication
1	**Hard failures must be hard everywhere.**	If the verifier zeros a trajectory, the judge must also return zero for that condition. Never apply partial credit to a fatal structural error.
2	**Align the judge prompt to verifier constraints.**	Every binary constraint enforced by the verifier should appear verbatim in the judge prompt, with a scored negative example.
3	**Log structural sub-metrics as first-class signals.**	Track each constraint-violation rate independently. Set alerts on violation rates, not only on aggregate reward.
4	**Gate the judge signal on verifier pass.**	Apply $r_j = 0$ whenever the verifier assigns $r = 0$ to prevent gradient leaks from judge partial credit.
5	**Structural constraints belong in the verifier, not the judge.**	RL policies transfer styles associated with high reward. If a structural property can be enforced exactly by a rule, enforce it as a hard verifier check rather than as a soft judge preference.

C LLM-as-Judge Prompt (v5.0)

We reproduce below the complete system prompt supplied to the LLM-as-judge component of our build-task reward pipeline (Section 3.1). The judge receives each generated tool (a Python function together with its OpenAI-compatible JSON schema) and returns a structured JSON object with five numeric quality scores and three structured text fields. The prompt is presented verbatim to support reproducibility.

LLM-as-Judge Prompt (v5.0)

You are a code reviewer evaluating Python tool implementations and their OpenAI function-calling schemas. These tools are used by LLMs to solve mathematical and reasoning tasks via tool-calling.

Important: what to look for

Do **not** attempt to mentally execute the code or verify that it produces correct numeric outputs; that approach is unreliable. Instead, assess the tool based on *structural* signals you can directly observe in the code and schema.

Required analysis process

Complete these steps before scoring.

Step 1: Check code structure for red flags. Look for these specific problems (list every one you find):

- **Incomplete implementation:** `pass`, `TODO`, `NotImplementedError`, empty branches, functions that only handle a subset of cases (e.g. only quadratic when the task is general polynomial).
- **Fragile parsing:** regex-based math parsing instead of proper libraries (`sympy`, `numpy`, `ast`); hardcoded patterns that won't generalise.
- **Wrong approach:** algorithm doesn't match the task type (e.g. brute-force search for a task that needs symbolic math).
- **Missing error paths:** bare `except: pass` that silently swallows errors; returning empty/`None` on failure without indication.
- **Hardcoded limits:** magic numbers, fixed-size assumptions, only works for specific input dimensions.
- **Truncated code:** function ends abruptly, code was cut off mid-implementation.

Step 2: Check if the code covers the task. Compare the task type and sample questions against what the code actually implements. A tool that handles only linear equations when Q1 is a cubic is fundamentally inadequate, regardless of how clean the code looks.

Step 3: Schema–code comparison. Compare schema parameter names, types, and descriptions against the function signature. Note every mismatch.

Step 4: Assess tool API design. If an LLM only sees the schema (not the code), could it construct correct function calls? Are parameter names and descriptions clear enough?

Scored examples

Study these examples carefully. They show common pitfalls, especially tools that *look* polished but are structurally broken.

Example A: Good tool (high scores).
Task: `count_bits` **Q1:** "How many 1 bits are in the binary representation of 76,778,227?"
Expected: 14

```
def count_one_bits(n):
    return bin(n).count('1')
```

Schema:

```
{
  "name": "count_one_bits",
  "parameters": {
    "properties": {
      "n": {
        "type": "integer",
        "description": "The non-negative integer whose binary
          representation is to be analyzed for the count of 1
             bits."
      }
    },
    "required": ["n"]
  }
}
```

Correct scores:

```
{
  "red_flags": "none",
  "task_coverage": "yes -- bin().count() handles any non-
     negative integer",
  "code_correctness": 4, "code_clarity": 4,
  "schema_quality": 4, "schema_code_alignment": 5, "
     overall_quality": 4
}
```

Why: Simple, correct approach using a reliable built-in that generalises to all inputs. Schema matches code exactly. Not 5 because there is no input validation (negative numbers give wrong results).

Example B: Polished but broken tool (low scores despite good appearance).
Task: `polynomial_equations` **Q1:** "Solve `q**5 + 21*q**4 + 44*q**3 + 45 = 0`"
Expected: $-18.6398,\ -2.0423,\ -1.3908$

```
import math

def solve_quadratic_equation(a, b, c):
    """Solves␣a*q^2␣+␣b*q␣+␣c␣=␣0␣and␣returns␣real␣decimal␣
        solutions."""
    if abs(a) < 1e-10:
        if abs(b) < 1e-10:
            return "0.0" if abs(c) < 1e-10 else ""
        return f"{-c/b:.4f}"
    discriminant = b**2 - 4*a*c
    if discriminant < 0:
        return ""
    ...
```

Schema:

```
{
  "name": "solve_quadratic_equation",
  "parameters": {
    "properties": {
      "a": {"type": "number"},
      "b": {"type": "number"},
      "c": {"type": "number"}
    },
    "required": ["a", "b", "c"]
  }
}
```

Correct scores:

```
{
  "red_flags": "function only handles quadratic (degree 2)
      equations
    but Q1 is degree 5; hardcoded to 3 coefficients",
  "task_coverage": "no -- cannot solve any polynomial above
      degree 2,
    which is the primary task requirement",
  "code_correctness": 1, "code_clarity": 4,
  "schema_quality": 4, "schema_code_alignment": 2, "
      overall_quality": 1
}
```

Why: The code *looks* clean (good naming, typed parameters, proper docstring) but it fundamentally cannot solve the task: it handles only degree-2 polynomials while the task requires general polynomial solving. `schema_code_alignment` is 2 because the schema doesn't warn users that it only handles quadratics.

Example C: Ugly but functional tool (moderate scores).
Task: `polynomial_equations` **Q1:** "Solve `q**5 + 21*q**4 + 44*q**3 + 45 = 0`"
Expected: $-18.6398,\ -2.0423,\ -1.3908$

```
def solve_equation(equation_str):
    from sympy import symbols, solve, Eq, sympify
    q = symbols('q')
    expr = sympify(equation_str.replace('=␣0', '').strip())
    solutions = solve(Eq(expr, 0), q)
```

```
    real_sols = [complex(s).real for s in solutions
                 if abs(complex(s).imag) < 1e-6]
    return ', '.join(f'{s:.4f}' for s in sorted(real_sols))
```

Schema:

```
{
  "name": "solve_equation",
  "parameters": {
    "properties": {
      "equation_str": {
        "type": "string",
        "description": "The equation to solve"
      }
    },
    "required": ["equation_str"]
  }
}
```

Correct scores:

```
{
  "red_flags": "uses sympify on raw string input (fragile);
     schema
    description is vague; single string argument instead of
       structured
    input",
  "task_coverage": "yes -- sympy.solve handles arbitrary
     polynomial
    degrees",
  "code_correctness": 3, "code_clarity": 1,
  "schema_quality": 1, "schema_code_alignment": 4, "
     overall_quality": 2
}
```

Why: The approach is correct (`sympy.solve` handles arbitrary polynomials), but the API is weak: one opaque string argument and a vague schema description. An LLM might pass the equation in a format `sympify` can't parse. `code_correctness` is 3 (not higher) because `sympify` on raw strings is fragile. Overall 2: despite API flaws the core algorithm works.

Key lesson from the examples

Example B is the trap to avoid: **do not** give high scores just because code is well-structured, well-named, or has good parameter decomposition. If the approach cannot handle the task (Step 2 fails), `code_correctness` **must** be 0–2 regardless of code quality. Example C shows the reverse: ugly code with a correct approach deserves higher `code_correctness` than polished code with a wrong approach.

Scoring rubrics (0–5)

`code_correctness` (0–5): *structural soundness: does the code implement a viable approach for the task?*

0 No implementation, syntax errors, or completely unrunnable.
1 Critical structural flaws: incomplete branches, truncated implementation, or approach fundamentally wrong for the task type.
2 Approach is plausible but has significant gaps: only handles a subset of cases, fragile parsing, or swallows errors silently.
3 Solid implementation with minor structural concerns (e.g. no input validation, hardcoded limits that might not cover all cases).
4 Clean implementation using appropriate libraries/algorithms; handles the task type fully.
5 Robust implementation with explicit error handling, input validation, and appropriate algorithm choice.

`code_clarity` (0–5): *naming quality and argument design for LLM usability*

0 No meaningful function: bare code snippet or hardcoded answer with no parameters.
1 Cryptic function name (e.g. `f`, `run`); all input is one opaque string argument.
2 Generic name (e.g. `solve`, `process`); arguments poorly named or bundled into one dict/string.
3 Name indicates purpose (e.g. `calculate_area`); arguments separated but could be decomposed further.
4 Descriptive name; arguments well-decomposed into typed parameters (e.g. `operator: str, a: float, b: float` instead of `expression: str`).
5 Self-documenting: precise function name; each argument is atomic, well-typed, and named so an LLM can call it without examples.

`schema_quality` (0–5): *OpenAI schema completeness and accuracy*

0 No schema or malformed JSON.
1 Missing required fields, wrong types, no descriptions.
2 Vague or misleading descriptions; types partially wrong.
3 Adequate descriptions, correct types, but missing format details or constraints.
4 Precise descriptions, correct types, required list correct.
5 Descriptions specify exact input format, constraints, value ranges, and examples.

`schema_code_alignment` (0–5): *does the schema accurately represent what the code does?*

0 Schema describes a completely different function.
1 Parameter names or types contradict the function signature.
2 Schema describes idealised behaviour the code doesn't implement.
3 Mostly aligned but schema over-promises (e.g. claims to handle cases the code skips).
4 Minor discrepancy only (e.g. one description slightly off).
5 Schema is a precise contract for the actual implementation.

`overall_quality` (0–5): *holistic: would an LLM get correct answers using this tool?*
Important: `overall_quality` **cannot exceed** `code_correctness` + 1. A tool with broken code is not redeemed by a nice schema.

0 Unusable: no working code or no schema.
1 Broken: critical structural flaws prevent reliable use.
2 Marginal: plausible approach but too many gaps for reliable results.
3 Functional: solid approach, works for the common case.
4 Good: correct approach, clear API, accurate schema.
5 Excellent: robust implementation with great API design.

Output format

Respond with **only** a JSON object:

```
{
  "red_flags":              "<list every structural problem
      found in Step 1,
                              or 'none' if clean>",
  "task_coverage":          "<yes/partially/no -- does the code
      cover the
                              task type and sample questions?>",
  "reasoning":              "<1-2 sentences justifying scores
      based on the
                              red flags and task coverage above
                                 >",
  "code_correctness":       <int 0-5>,
  "code_clarity":           <int 0-5>,
  "schema_quality":         <int 0-5>,
  "schema_code_alignment": <int 0-5>,
  "overall_quality":        <int 0-5>
}
```

D Build and Use Task Prompt Templates

This appendix documents the prompt templates used during SMITH training. All prompts are held fixed throughout training; no prompt engineering is done between training runs.

D.1 Build-Task Prompt

The build task follows a two-part message structure: a *system message* that establishes the model's role, and a *user message* that combines detailed tool-writing instructions with the $N{=}4$ in-context `(question, answer)` pairs.

System message.

Build-Task System Message

You are an expert tool builder.

User message structure. The user message concatenates three components in order:

1. **Tool-maker instruction block.** A fixed multi-paragraph prompt (abridged below) specifying:
   - *Core principles*: the model proposes, execution verifies ("Model Proposes, You Dispose"); parameters must be atomic and typed; function names must follow a Verb-Noun convention.
   - *Best practices*: use standard libraries (`sympy`, `numpy`) over fragile regex; handle edge cases explicitly; never hard-code input-specific values.
   - *Schema requirements*: the `name` field must match the top-level Python function name exactly; every parameter must have a `description`; the `required` array must list all mandatory parameters.
   - *Output format instruction*: the response must contain **exactly one** ```` ```python ```` block and **exactly one** ```` ```json ```` block; no other text is permitted.
2. **In-context examples.** $N{=}4$ question-answer pairs from the easy difficulty band of the current task category, formatted as:

   `Question 1: {q_1}\n Answer: {a_1}\n\n Question 2: {q_2}\n Answer: {a_2}\n\n` ...
3. **Hidden metadata.** A `<tool_rl_metadata>...</tool_rl_metadata>` block containing JSON-encoded task metadata (task category, test set, ground truth answers). This block is appended by the training harness and stripped before the prompt is sent to the policy, so the model *never* observes the ground-truth test answers during generation.

Tool constraints. Every build-task instance additionally specifies two naming constraints that are checked by the structural verifier: the generated function must be named `solve` and must accept a single parameter named `question`. These constraints are stated in the instruction block and enforced by the format reward r^{fmt}.

D.2 Use-Task Prompt

The use task presents the model with a tool (in JSON schema form) and asks it to invoke the tool to answer a question. The model sees only the OpenAI-compatible schema, never the underlying Python code, testing whether the schema is clear enough to drive correct invocation.

System message.

Use-Task System Message

You are a helpful assistant that can build tools and then use them.

Tool injection. The tool schemas available to the model are injected via the standard `tools` parameter of the OpenAI chat-completion API. Each use-task rollout receives up to $m{=}3$ schemas: 1 domain tool (from the correct category) and 2 distractor tools (from randomly selected other

categories). This forces the model to identify and invoke the correct schema rather than calling the first available tool by default.

User message. The user message contains only the target question:

Use-Task User Message (template)

Solve the following question. When you have the final answer, present it as:\n \boxed{your answer here}\n\n {question text}

Multi-turn dialogue. The model responds with an OpenAI `tool_calls` message specifying which schema to invoke and with what arguments. The training harness executes the Python function, formats the return value as a `role: tool` message, and appends it to the conversation. This continues for up to $T=5$ turns; the final answer must be presented inside a `\boxed{}` delimiter.

D.3 Evaluator Prompt for r^{eval}

The evaluator model π^{eval} (Sec. 3.1) answers each of the $K=16$ held-out test questions using the generated tool, following the same use-task prompt structure described above. Crucially:

- **Only the JSON schema** is shown; the evaluator never receives the Python implementation.
- **Correctness** is verified by an LLM equivalence judge (Section 3.1) that compares the evaluator's final answer with the ground truth; exact string matching is used as a fast-path fallback.
- **Counting rule**: a question counts as correct only if the evaluator's final answer was produced via a successful tool call. A text-only answer (without invoking the tool) does not contribute to r^{eval}, preventing the evaluator from exploiting its own reasoning ability to bypass the tool-use objective.
- **LoRA synchronisation**: π^{eval} is periodically refreshed by copying the latest training checkpoint weights (every 5 gradient steps), providing an improving evaluation target as the policy becomes a better tool user. See Appendix B for a detailed discussion of the synchronisation artefacts this can introduce.

E Tool Pool Design

The Tool Pool $\mathcal{P}$ is a lightweight caching mechanism that decouples tool creation from tool use within a single training batch. Once a build-task rollout produces a valid tool, subsequent use-task rollouts in the same and future batches can invoke it directly without re-running the build step. This section documents the pool's internal structure, admission criteria, eviction policy, distractor selection, and initialisation.

E.1 Pool Structure

$\mathcal{P}$ is a thread-safe dictionary keyed by *task category*:

$$\mathcal{P} : \text{category} \;\rightarrow\; \big[\,\{\texttt{python_code},\ \texttt{openai_tools},\ \texttt{quality}\}\,\big]$$

Each entry stores the raw Python function string, the list of OpenAI-compatible tool schema dicts, and the evaluation quality score r^{eval} at the time of admission. Access is protected by a single re-entrant lock, allowing concurrent rollout workers to read and write without data races.

The task category of each rollout is parsed from the dataset row identifier (e.g. the key `"bitwise_arithmetic-train-build-42"` maps to category `"bitwise_arithmetic"`).

E.2 Admission Criterion

A tool generated during a build-task rollout is admitted to the pool if and only if $r^{\text{eval}} > 0$, i.e. the tool answers at least one of the $K=16$ held-out test questions correctly. Tools with $r^{\text{eval}} = 0$ (including format failures and execution errors) are discarded and never cached.

E.3 Capacity Cap and Eviction Policy

Each category bucket is capped at $C{=}20$ entries. When a new tool is admitted and the bucket is already full, the entry with the **lowest** r^{eval} score is evicted to make room. If multiple entries share the same minimum score, the **oldest** entry (i.e. the one inserted earliest) is chosen as the tiebreaker, implementing a "quality-first, recency-as-tiebreaker" policy.

This eviction policy has a natural curriculum effect: as training advances and the policy writes better tools, the pool's minimum quality threshold rises organically. Use-task rollouts in later training steps therefore face a higher-quality and more competitive pool than in the early steps, providing an implicit difficulty curriculum without any explicit scheduling.

E.4 Retrieval and Distractor Selection

When a use-task rollout arrives for category k:

1. **Domain tools.** Up to $m_d{=}1$ tool is retrieved from $\mathcal{P}[k]$, taken from the *most recently admitted* entries (last inserted). If $\mathcal{P}[k]$ is empty, the use-task rollout performs a fresh build pass first.
2. **Distractor tools.** Up to $m_{\text{dist}}{=}2$ tools are drawn from categories $k' \neq k$ with non-empty buckets. One tool is sampled uniformly at random from each eligible category, and the selected categories are shuffled before injection, maximising diversity across rollouts.
3. **Prompt injection.** The domain tool and distractor tools are concatenated into a single `tools` list and passed to the OpenAI API, exactly as if the model had built all three itself. The model must identify the correct tool by schema inspection and invoke it with the right arguments.

The combination of 1 domain tool and 2 distractors means the model cannot succeed by calling the first tool at random: it must parse the schemas, identify which function is relevant to the question, and construct a valid argument dict.

E.5 Initialisation

The pool starts **empty** at the beginning of training. In the first training steps, all use-task rollouts that require a category not yet covered by the pool must first perform a build pass. The 1:1 ratio of build and use tasks in each batch (Sec. 3.1) ensures that build tasks fire frequently enough in the early steps to populate the pool quickly.

Once a category bucket crosses the threshold of at least one valid tool, subsequent use-task rollouts in that category switch to the pool-retrieval path and skip the build step, saving approximately one LLM forward pass per affected rollout.

F Full Easy-to-Hard Difficulty Split for All 13 Training Tasks

Section 4.4 describes the easy-to-hard training protocol and gives a detailed example for the `cryptarithm` task. This appendix provides the complete mapping for all 13 training categories: the induction difficulty band (from which the $N{=}4$ in-context examples are drawn) and the evaluation difficulty band (against which r^{eval} is computed).

The difficulty bands are defined per-task by progressively harder instantiation parameters in Reasoning-Gym's procedural generator. Band labels follow the generator's internal scale: higher numbers correspond to harder instances. For every task, the induction context uses the *easiest* band (Band 2) and the evaluation set uses the *hardest* available band. This separation ensures that a tool which merely memorises the induction examples scores zero on the evaluation set; only a tool that encodes the underlying algorithm generalises across the gap.

Training and evaluation set sizes. For each task, the training set (induction band) consists of 500 instances at difficulty Band 2. For tasks whose easy-to-hard gap is large enough to warrant it, an additional 500 instances from Band 3 (medium) are included in training, though the evaluation set always draws from the hardest available band. The evaluation set used to compute r^{eval} at each rollout step consists of $K{=}16$ instances sampled at inference time from the hardest difficulty band.

Table 11: Induction context and evaluation difficulty bands for all 13 SMITH training tasks. *Induction band* defines the difficulty of examples shown in the $N{=}4$ build-task context. *Evaluation band* defines the difficulty of instances used to compute r^{eval}. *Key difficulty axis* is the parameter that grows from easy to hard.

Task	Category	Induction band (easy)	Evaluation band (hard)	Key difficulty axis
Bitwise arithmetic	Arithmetic	Expression depth 2	Expression depth 4–5	Nesting depth of the bit-operation expression tree
Cryptarithmetic	Arithmetic	$\leq$ 8 unique letters (easy/medium puzzles)	$\geq$ 9 unique letters, including 10-letter puzzles such as `FORTY+TEN+TEN=SIXTY`	Number of unique letters (exponential search space growth)
Bit counting	Algorithms	Integer values 1–10^6	Integer values 10^7–10^8	Magnitude of the input integer
LCM	Algorithms	2 numbers, values 1–50	3–4 numbers, values 100–500	Count and magnitude of operands
GCD	Algorithms	2 numbers, values 1–500	3–4 numbers, values 1,000–5,000	Count and magnitude of operands
Base conversion	Algorithms	Bases 2–10, values 1–500	Bases 2–16, values 2,000–5,000	Target base range and magnitude of the number
Isomorphic string	Algorithms	String length 10–19	String length 31–40	String length (longer strings require tracking more character mappings)
Polynomial equations	Algebra	2–3 terms, degree 1–2, coefficients 1–10	5–6 terms, degree 3–5, coefficients 1–50	Polynomial degree and number of terms
Polynomial multiplication	Algebra	2–3 terms per polynomial, degree 1–2, 2 polynomials, coefficients 1–5	5–6 terms, degree 3–5, 2–3 polynomials, coefficients 1–12	Degree, term count, and number of polynomials to multiply
Countdown	Games	4 numbers, target 10–100	6 numbers, target 10–200	Number of available operands and target range
Tower of Hanoi	Games	3 disks (7 optimal moves)	5 disks (31 optimal moves)	Number of disks (solution length grows as $2^n - 1$)
Knights and Knaves	Logic	2 characters, depth 2, width 3	4 characters, depth 4, width 5	Number of characters and depth of logical deduction tree
Caesar cipher	Logic	3–10 words, rotation 1–10	12–20 words, rotation 15–25	Text length and rotation offset (larger offsets are less guessable)

Final benchmark accuracy (reported in Table 1) is measured over 70 instances per difficulty level, stratified uniformly across all bands.

Task-category mapping. The 13 training categories span five higher-level groups: *Arithmetic* (*bitwise arithmetic*, *cryptarithmetic*), *Algorithms* (*bit counting*, *LCM*, *GCD*, *base conversion*, *isomorphic string*), *Algebra* (*polynomial equations*, *polynomial multiplication*), *Games* (*countdown*, *Tower of Hanoi*), and *Logic* (*knights and knaves*, *Caesar cipher*). This spread ensures that the RL policy is exposed to a variety of algorithm types, preventing the easy-to-hard protocol from specialising to a single reasoning pattern.

G OOD Inference Protocol for TabMWP-Hard and GQA

Section 4.2 describes the two out-of-domain benchmarks but does not detail the exact inference procedure. This appendix documents the protocol used for TabMWP-Hard and GQA, which mirrors the Reasoning-Gym evaluation loop as closely as possible so that differences in performance can be attributed to domain shift rather than to evaluation asymmetry.

G.1 Single Build-Then-Use Loop

At evaluation time, for every benchmark the model performs a single *build* pass, then applies the resulting tool to every test instance:

1. **Build pass.** A small set of reference `(question, answer)` pairs is sampled from the benchmark's own training (or validation) split and presented to the model using the standard build-task prompt (Appendix D). The model generates a single Python function and a matching OpenAI-compatible JSON schema in one forward pass.
2. **No retry.** If the build pass fails (syntax error, schema–code mismatch, or execution error), the tool is discarded and every test instance in that run receives a score of zero. No additional build attempts are made.
3. **Use pass.** Each test question is presented to the model together with the generated tool's JSON schema in standard OpenAI function-calling format. The model may issue up to $T{=}5$ tool calls before producing a final answer. A test question scores 1 if the model invokes the tool and the returned answer matches the ground truth, and 0 otherwise.

This "single-shot build-then-use" structure is identical to the Reasoning-Gym evaluation loop used for RG (Seen) and RG (Unseen): the same tool that was induced from easy in-context examples is applied without modification to all test instances.

G.2 In-Context Examples

In-context examples for the build pass are drawn from each benchmark's own training split, ensuring the model has a domain-appropriate induction context:

TabMWP-Hard. We sample $N{=}4$ `(question, table, answer)` triples from the original TabMWP training split (before augmentation). The table is serialised as a plain-text pipe-delimited string and prepended to the question text, matching the format of the test instances. The same examples are used for every evaluation run.

GQA. We sample $N{=}10$ `(question, answer)` pairs from the GQA training split. As described in Appendix H, no image is shown in the build context; the model instead sees only the question strings and must infer how to compose the visual primitives to answer them. A larger in-context set ($N{=}10$ vs. $N{=}4$) is used for GQA because the diversity of visual question types is higher and more examples are needed for the model to infer a general strategy.

G.3 Protocol Differences from the RG Training Loop

Three differences distinguish the OOD evaluation protocol from the training-time Reasoning-Gym loop:

1. **Easy-to-hard gap is absent.** For Reasoning-Gym, the induction context is drawn from easy curriculum bands while the evaluation set is drawn from hard bands (Sec. 4.4). For OOD benchmarks there is no such difficulty stratification; in-context examples and test instances are drawn from the same distribution.
2. **Tool Pool is not used.** At evaluation time, no pre-built tools from the training Tool Pool are injected. The model always performs a fresh build from the provided in-context examples.
3. **Evaluation reward (r^{eval}) is not computed.** r^{eval} is a training-time signal used to grade tools against a held-out set of RG instances. At OOD evaluation time, the only signal is final test-split accuracy.

Together, these differences mean that OOD performance reflects the model's ability to generalise its *tool-writing strategy* to new domains, not its ability to exploit training-distribution shortcuts.

H GQA Visual Tool Setup

Because SMITH is trained exclusively on text-based Reasoning-Gym tasks, the model has no direct perception ability at evaluation time. To enable it to answer GQA questions nonetheless, we provide a fixed set of three *visual primitive* functions that are pre-implemented and served via a local API. The model's job during the build task is to compose these primitives into a reusable tool; it never has to implement low-level vision code itself.

H.1 Visual Primitive Functions

Three primitives are available to the model:

1. `locate_objects(image_b64, object_name)`: Uses OWL-ViT [35] (`owlvit-base-patch16`) to perform open-vocabulary object detection. Returns a list of bounding boxes in $[\,x_1, y_1, x_2, y_2\,]$ format for all detected instances of `object_name` within the base-64-encoded JPEG image `image_b64`.
2. `visual_qa(image_b64, question)`: Uses BLIP-VQA [36] (`Salesforce/blip-vqa-base`) to answer a free-form natural language `question` about the image. Returns a short free-text answer string.
3. `crop_region(image_b64, boxes)`: A pure-Python utility (no neural model) that crops the image to the first bounding box in `boxes`, with a $1.5\times$ padding margin, and returns the cropped region as a new base-64 JPEG string.

All three primitives are served by a FastAPI server at `localhost:8000` (configurable via the `$GQA_SERVER_URL` environment variable).

H.2 Image Encoding

GQA test images are loaded from the HuggingFace dataset `[anonymous]/gqa-testdev-balanced`. Before each tool invocation, the PIL image for the current question is encoded as a base-64 JPEG string and bound to the Python global variable `IMAGE`. The generated tool code always reads from this global; it does not accept an image path or URL argument. This design keeps the tool interface simple (`question: str` $\rightarrow$ `str`) while still providing image access:

```
# Tool interface (fixed across all GQA tools)
def solve(question: str) -> str:
    # IMAGE is pre-loaded as a global b64-encoded JPEG
    objects = locate_objects(IMAGE, ...)
    answer  = visual_qa(IMAGE, question)
    return answer
```

H.3 Build-Task Prompt for GQA

Before the standard tool-maker instruction block, the build-task prompt is prepended with a *primitives context* block that (a) declares the three available functions with their full signatures and docstrings, and (b) instructs the model that it must compose these primitives rather than re-implementing vision logic. The key constraint injected into the prompt is:

> *"Your generated tool must accept only `question: str`, compose the primitives above as needed, and return the answer as a plain string. Do not re-implement `locate_objects`, `visual_qa`, or `crop_region`; call them directly."*

The $N{=}10$ in-context examples shown to the model during the build task are sampled uniformly at random from the GQA training split. Each example is a `(question, answer)` pair; no image is shown in the build-task prompt itself, so the model must reason about image content solely through the lens of the available primitives.

H.4 Tool Selection and Evaluation

Because visual tool generation is noisier than text-only tool generation (the model has no direct visual feedback during tool writing), we generate $M=10$ candidate tool proposals and select the best-performing one:

1. Sample $N=10$ reference `(question, answer)` pairs from the GQA training split as the in-context build context.
2. Generate $M=10$ candidate Python tools using the build-task prompt.
3. Evaluate each candidate on a disjoint validation set of 100 questions drawn from the GQA training split; record per-candidate validation accuracy.
4. Select the candidate with the highest validation accuracy as the final tool.
5. Apply the selected tool to the full GQA test split; report test-split accuracy.

A failed build (syntax error or schema mismatch) receives validation accuracy zero and is never selected unless all M candidates fail, in which case a fallback empty-response is returned and every test question scores zero.

H.5 Answer Verification

GQA answers are short free-form strings (e.g. "yes", "blue", "3"). We use exact string match after lowercasing and whitespace stripping. No LLM equivalence judge is applied; the simplicity of GQA answers makes rule-based exact match sufficient and avoids judge overhead at the scale of 2,516 test questions.

I ReTool Evaluation Setup

ReTool [24] is included as a distillation baseline to isolate the contribution of the schema-grounded tool representation: both ReTool and SMITH use execution feedback, but only SMITH produces reusable callable schemas. This appendix documents the specific checkpoint, training configuration, and evaluation protocol used for our ReTool results.

I.1 Checkpoint and Training Configuration

We fine-tune QWEN3-4B-INSTRUCT on the official ReTool-SFT dataset (`JoeYing/ReTool-SFT` on HuggingFace), which contains code-execution trajectories distilled from QWEN-32B. Training uses QLoRA with the following hyperparameters:

Table 12: ReTool fine-tuning hyperparameters.

Hyperparameter	Value
Base model	QWEN3-4B-INSTRUCT
Training dataset	`JoeYing/ReTool-SFT`
Adapter	QLoRA (4-bit NF4, double quantisation)
LoRA rank r	64
LoRA α	128
LoRA dropout	0.05
LoRA target modules	all linear layers
Epochs	6
Sequence length	5,120 tokens
Gradient accumulation	8
Micro-batch size	2
Optimizer	`paged_adamw_32bit`
LR schedule	Cosine
Learning rate	4×10^{-5}
Warmup fraction	0.1

This matches the LoRA configuration used for SMITH (r=64, α=128), so that any performance difference between ReTool and SMITH is attributable to the training objective rather than the adapter capacity.

I.2 Evaluation Protocol on Reasoning-Gym and OOD Benchmarks

Code execution. ReTool operates in a multi-turn loop: the model generates code in <code>‘‘‘python...
‘‘‘</code> blocks, a sandbox executes each block and returns output wrapped in <interpreter>output</interpreter> tags, and the model continues reasoning from the execution trace. We allow up to T_{rt}=10 code-execution rounds per question. A question terminates when the model emits an <answer>\boxed{...}</answer> block, or when the turn budget is exhausted. The final answer is extracted from the \boxed{} delimiters inside the <answer> tag.

Reasoning-Gym and TabMWP-Hard. Each test question is presented directly to the ReTool model with its standard system prompt; no tool schema or in-context examples are provided. Because ReTool generates per-question code rather than a reusable schema, there is no build pass or tool pool: every question triggers a fresh code-generation episode.

GQA adaptation. Answering GQA questions requires access to visual primitives. We adapt ReTool for GQA by (a) augmenting the system prompt with descriptions of the three visual primitive functions (`locate_objects`, `visual_qa`, `crop_region`; see Appendix H) and (b) prepending the primitive implementation code to every sandbox execution block so that `import`-free calls succeed. With these additions, ReTool can incorporate vision into its reasoning chains via direct function calls, on equal footing with SMITH's composed tool.

I.3 Key Distinction from SMITH

ReTool and SMITH both use execution feedback, but differ in two structural respects that are the central comparison axis of this work:

1. **Reusability.** ReTool generates ephemeral code per question; the code is discarded after each turn. SMITH generates a callable JSON schema, a structured interface that can be stored in the Tool Pool and reused across many questions without re-running the build step.
2. **Training signal.** ReTool is trained by behavioural cloning from a 32B oracle. SMITH is trained end-to-end from a verifiable reward, with no strong oracle required at training time.

ReTool achieves the highest in-distribution accuracy on RG (Seen) (92.0 vs. our 86.6), reflecting the advantage of a 32B distillation oracle on seen task families. SMITH's advantage emerges on held-out transfer (RG Unseen, GQA, TabMWP-Hard), where the reusable schema and RL-trained generalisation provide a structural edge over per-question code generation.

J LATM (distill GPT-4.1) Baseline Setup

The **LATM (4B distill GPT-4.1)** baseline tests whether behavioural cloning from a strong frozen oracle can match RL training over an explicit reward signal. The oracle is GPT-4.1, which generates tool-writing trajectories for each of the 13 training task categories; a Qwen3-4B model is then fine-tuned on these trajectories. This appendix documents the data collection pipeline, the fine-tuning configuration, and the evaluation protocol.

J.1 Trajectory Collection

Oracle model. Tool-writing trajectories are generated by GPT-4.1 via the OpenAI API. For each task category, GPT-4.1 is given the same build-task instruction block used in SMITH (Appendix D), together with N=5 question-answer pairs sampled from the easy difficulty band.

Trajectory volume. For each task category, we collect M=8 generation attempts per skill-set sample, where a *skill set* is a particular draw of N in-context examples. Both tool-making trajectories

(oracle response = Python code + JSON schema) and tool-use rollouts (multi-turn conversations where the tool is invoked successfully to produce a correct answer) are retained.

Tool-use rollout collection. After tool generation, a frozen QWEN3-4B-INSTRUCT model performs up to 10 tool-call rounds for each training question using the oracle-generated schema. Multi-turn conversations are retained only when (a) the tool call executed without error and (b) the produced answer matched the ground truth. Up to 1,000 such conversations are collected per task category to balance dataset size across categories.

Distractor augmentation. To teach the student model to select the correct tool among multiple candidates, each retained tool-use conversation is augmented with distractor schemas at two rates: 50% of examples receive one distractor (from a randomly chosen different category) and 90% receive a second distractor. Augmentation applies the same pool-retrieval logic as SMITH (Appendix E), ensuring comparability.

The final dataset contains tool-making conversations and distractor-augmented tool-use conversations for all 13 training categories.

J.2 Fine-Tuning Configuration

We fine-tune QWEN3-4B-INSTRUCT on the collected SFT dataset using full-parameter fine-tuning (no LoRA adapter). The training hyperparameters are:

Table 13: LATM (distill GPT-4.1) fine-tuning hyperparameters.

Hyperparameter	Value
Base model	QWEN3-4B-INSTRUCT
Adapter	None (full fine-tuning)
Epochs	4
Sequence length	16,000 tokens
Gradient accumulation	18
Micro-batch size	2
Optimizer	`adamw_torch`
LR schedule	Cosine
Learning rate	4×10^{-5}
Warmup fraction	0.1
Training format	Chat template; train on assistant turns only

Fine-tuning uses full-parameter updates (not LoRA), in contrast to SMITH's LoRA configuration (r=64, α=128). This means the LATM distilled model modifies all model weights and may have a larger effective capacity for memorising the oracle's style. The comparison therefore tests *training signal quality* (RL reward vs. behavioural cloning) rather than model capacity.

J.3 Evaluation Protocol

At evaluation time, the fine-tuned LATM model is evaluated using the standard build-then-use inference protocol (Appendix G). No oracle model is involved at test time; the student model generates tools autonomously from the N=4 in-context examples. Results are reported across RG (Seen), RG (Unseen), TabMWP-Hard, and GQA in Table 1 and Table 4.

J.4 Comparison to SMITH

The key structural difference between LATM (distill GPT-4.1) and SMITH is the source of the training signal:

- **LATM (distill GPT-4.1)** trains by imitating GPT-4.1 outputs. The student learns to reproduce what GPT-4.1 writes, regardless of whether those tools actually execute correctly on held-out instances.

- **SMITH** trains from a verifiable execution reward. Every gradient update is conditioned on whether the generated tool answered the held-out test questions correctly; no oracle demonstrations are required.

LATM (distill GPT-4.1) achieves strong in-distribution accuracy (83.3 on RG Seen) because GPT-4.1 generates high-quality tools for the training task categories. SMITH's advantage emerges on transfer benchmarks (RG Unseen 79.8 vs. 66.4; GQA 42.6 vs. $\mathbf{56.0}^{\dagger}$), where the execution reward drives the policy toward tools that generalise beyond the oracle's demonstrated style.

†The GPT-4.1 distillation gap on GQA reflects that GPT-4.1 has native visual understanding and generates visual primitives the 4B student can imitate; SMITH reaches its GQA score entirely through the self-supervised tool-creation loop without visual oracle data.

K Agentic Loop Comparison: KTCE, CRAFT, TroVE, and SMITH

This appendix provides a detailed, implementation-level comparison of the agentic designs of the three baseline systems evaluated in Section 4.5 (KTCE [12], CRAFT [10], and TroVE [11]) alongside our own SMITH framework. The central axis of variation is *when* tool creation happens relative to test-time inference.

K.1 Positioning Overview

Table 14 situates each system along the tool-creation timeline before discussing the per-system loops in detail.

Table 14: Tool-creation timing, model modification, and cross-problem reuse for each system.

System	Tool creation timing	Weights changed?	Tools survive across problems?
KTCE	Fully offline (before any test problem)	No	Yes (fixed toolset)
CRAFT	Fully offline (before any test problem)	No	Yes (fixed library)
TroVE	Online / streaming (during test inference)	No	Yes (library grows sequentially)
SMITH	Trained offline via RL	**Yes**	Yes (quality-filtered pool)

K.2 KTCE: Offline Evolutionary Creation with Online Retrieval-then-Solve

KTCE separates tool creation from inference with a hard boundary. All tool work happens before any test problem is seen.

Offline phase. Training problems are grouped by mathematical subfield using BGE-M3 embeddings and k-means clustering. For each subfield cluster, an LLM generates candidate Python functions; semantically near-duplicate candidates are collapsed via agglomerative clustering (similarity ≥ 0.80), and one verified tool per cluster is selected by majority-vote execution. This initial toolset then enters a 5-iteration evolutionary loop:

1. **Evaluate**: run all training problems in the subfield through the current toolset; record per-tool usage frequency (`Freq`) and tool success rate (`TSR`). Compute a composite loss $\alpha \sum Q_{\text{tool}} + \beta Q_{\text{set}} + \gamma \max(0, n - k)$.
2. **Delete**: LLM decides which tools (up to 3–5) to remove based on low frequency and success rate.
3. **Modify**: for each tool with TSR/Freq ≤ 0.90, generate an evolved version using failure examples as context; validate by execution before substituting.
4. **Add**: LLM proposes new tools for problems currently uncovered; validated by execution before insertion.
5. **Rollback**: if loss increases, revert to the previous iteration and pass failure context to the next modify/add step.

Each tool record carries a natural-language `experience_pool`: usage examples accumulated during the evaluate step. The final toolset is organised as a two-level map: Field $\rightarrow$ Subfield $\rightarrow$ [tool].

Online phase (per test problem).

1. Retrieve the subfield from per-problem metadata.
2. **LLM call 1**: model reads a numbered list of subfield tools and selects which to use; output parsed for tool indices.
3. **LLM call 2**: model generates Python code that calls the selected tools, augmented by BGE-M3-ranked few-shot examples from the experience pool. Code is executed; output (`hint`) is captured.
4. **LLM call 3**: chain-of-thought extraction of the final answer from the hint.

LLM calls at test time: 3 (retrieval select + code-gen + CoT extract).

K.3 CRAFT: Offline Diversity-Sampled Creation with Multi-View Retrieval

CRAFT constructs its tool library offline using GPT-4 for both creation and abstraction steps, then switches to GPT-3.5-turbo for inference.

Offline phase. Training problems are sampled in diversity-maximising epochs: epoch 0 draws 200 random problems; each subsequent epoch ranks remaining problems by their minimum SimCSE cosine similarity to already-sampled problems and takes the 100 most dissimilar. For each sampled problem:

1. **LLM call 1 (GPT-4)**: generate a specific Python solution for this problem. Execute and grade; discard if incorrect.
2. **LLM call 2 (GPT-4)**: abstract the specific solution into a general parametric function with a docstring. Execute the abstract tool; discard if non-executable.

Surviving tools are deduplicated by function-name similarity. Each tool is then encoded into four SimCSE vectors: (i) function name, (ii) one-line explanation, (iii) full docstring, (iv) the original problem that generated the tool. These are stored as `vector_library.pt` for retrieval.

Online phase (per test problem).

1. Encode the test question with SimCSE; score against all four vector views in the library; retrieve top-k tools ($k = 3 \rightarrow 2 \rightarrow 1$ if context overflows).
2. **LLM call 1**: question + retrieved tool code (pasted inline) $\rightarrow$ generate a new specific solution that may call or extend the retrieved functions.
3. **LLM call 2**: question + new solution $\rightarrow$ generate the API call expression.
4. Execute retrieved tools + new code + API call; grade against ground truth.

LLM calls at test time: 2 (code-gen + API call).

K.4 TroVE: Online Streaming Creation with Frequency-Based Forgetting

TroVE has no offline pre-processing phase. The library starts from a minimal default toolbox (standard imports) and grows as test problems are processed in order.

Online phase (per test problem, processed sequentially). Three modes are run in parallel for each problem:

1. **Import mode**: the full *current* library is injected into the prompt; the LLM generates a solution using existing functions.
2. **Create mode**: only the default (initial) library is shown; the LLM generates a solution *and* defines new helper functions it believes can be reused.
3. **Skip mode**: no library is shown; the LLM generates a plain Python solution.

Each mode's response is executed and graded. The best response (by correctness, then executability) is selected:

- If **import** wins: increment the usage-frequency counter for each called function.
- If **create** wins and execution succeeded: add the new functions to the library.

Periodic forgetting. Every 500 problems, tools whose frequency falls below $\log_{20}(n)$ (where n is the number of problems processed so far) are pruned from the library. Problems whose winning tool was pruned are queued for re-generation using only the import and skip modes after all problems have been processed.

LLM calls at test time: 3 per problem (one per mode, run in parallel). Later problems benefit from tools created for earlier ones; the result therefore depends on problem order.

K.5 SMITH: RL-Trained Tool Creation Coupled with Tool Use

SMITH does not prompt a frozen model to write tools at test time. Instead, it trains a 4B model via RL to be a capable tool creator, jointly optimising tool creation and tool use inside a single learning objective.

Training loop. Training alternates between two task types on 13 procedural task categories from Reasoning-Gym:

- **Build tasks**: given a task description and a small set of in-context examples, generate a Python function and a matching OpenAI-compatible JSON schema in a single forward pass. Three independent reward signals are computed: execution accuracy on held-out questions (r_{eval}), LLM-as-judge code quality (r_{judge}, see Appendix C), and format consistency of the JSON schema (r_{format}). Correctness reward is granted *only* when the final answer is produced through a successful tool invocation, removing any incentive to substitute text-only reasoning.

- **Use tasks**: given a tool from the shared pool, invoke it correctly to answer a held-out question; reward is rule-based answer matching.

The shared pool admits tools only after they pass execution evaluation. As the pool fills, weaker tools are evicted, creating an implicit curriculum that exposes the policy to progressively stronger competition.

Online phase (per test problem).

1. **Build**: 1 LLM pass generates the Python function and JSON schema.
2. **Use**: LLM invokes the tool via the schema; execution returns the result; LLM synthesises the final answer.

LLM calls at test time: $\sim$**3** (build + invoke + answer synthesis). Unlike all baselines, the model has been *trained* to write the kind of tool it can also reliably invoke.

K.6 Full Comparison Table

Table 15 summarises all dimensions across the four systems.

Table 15: Implementation-level comparison of agentic tool-creation designs.

Dimension	KTCE	CRAFT	TroVE	SMITH
Tool creation trigger	Per knowledge-subfield cluster of training data	Per training problem (diversity-sampled across epochs)	Per test problem that yields a novel, executable function	RL training rollout on build tasks
Tool structure	Python function + name / docstring / `experience_pool`	Python function + docstring + 4 SimCSE embedding vectors	Python function + docstring + frequency counter	Python function + OpenAI JSON schema
Tool verification	Execution + majority-vote during creation; loss-tracked during optimization	Execution + answer correctness at both specific and abstract stages	Implicit: tool enters library only if full solution executes correctly and wins 3-way selection	Execution accuracy as RL reward; format consistency as separate reward axis
Optimization / refinement	Explicit 5-iteration evolutionary loop: evaluate $\rightarrow$ compute loss $\rightarrow$ LLM delete / modify / add $\rightarrow$ rollback if worse	None (one-shot: create, validate, deduplicate)	Implicit frequency-based forgetting: low-reuse tools pruned every 500 examples	RL training is the optimization loop; gradient updates improve the tool-writing policy itself
Retrieval mechanism	Two-stage: (1) subfield lookup from metadata; (2) LLM reads numbered list and selects tools	Multi-view SimCSE similarity across 4 views (name, explanation, docstring, original question)	None; entire current library injected into the import-mode prompt, library size bounded by trimming	None at test time; model generates the required tool directly (trained to do so)
Tool reuse across problems	Yes (fixed toolset shared across all test problems)	Yes (fixed library shared across all test problems)	Yes (tools created for problem i available for problem $i{+}1$ onward)	Yes (shared execution-verified pool)
Order dependency	No (offline toolset is order-independent)	No (offline library is order-independent)	**Yes**; later problems benefit from tools created for earlier ones, and shuffling changes results	No at inference (pool is pre-built)

Table 15 (continued)

Dimension	KTCE	CRAFT	TroVE	SMITH
Agent loop at inference	Linear: subfield lookup → LLM selects tools → LLM generates code → execute → LLM CoT extract	Linear: multi-view retrieve → LLM generates code (may extend tool) → LLM generates API call → execute	3-way parallel per problem (import / create / skip) → select best → conditionally update library	Build: 1-pass generation of function + schema; Use: invoke → execute → synthesise answer
LLM calls at test time	3 (retrieval select + code-gen + CoT extract)	2 (code-gen + API call)	3 (one per parallel mode)	∼3 (build + invoke + answer synthesis)
LLM calls during tool creation	100s–1000s total (5 iters × N problems × multiple calls per iter, per subfield)	2 per training sample (specific solution + abstraction), both GPT-4	0 (no offline phase)	RL training rollouts (amortised into model weights)
Semantic embeddings	BGE-M3 (clustering, dedup, few-shot retrieval)	SimCSE (diversity sampling + multi-view retrieval)	None	None
Training required?	No	No	No	**Yes** (RL fine-tuning)
Primary model(s)	GPT-3.5-turbo (retrieval, solve, evolve); BGE-M3 for embeddings	GPT-4 (construction); GPT-3.5-turbo (inference); SimCSE (retrieval)	CodeLlama-7b (default) or any OpenAI-compatible model	4B RL-trained model

K.7 Key Conceptual Distinctions

What the LLM sees at inference. The four systems differ fundamentally in how a tool is *presented* to the model at solve time:

- **KTCE** pastes tool code, docstring, and experience-pool examples inline into the solution prompt; the LLM writes new code that calls the tool functions directly.
- **CRAFT** pastes retrieved tool code inline; the LLM writes new code that may call or extend the retrieved functions.
- **TroVE** pastes the *entire current library* of function definitions into the import-mode prompt; the LLM writes code that imports from the toolbox by name.
- **SMITH** presents the tool as an OpenAI function-calling schema (not raw code); the LLM issues a `tool_calls` message, the Python function is invoked externally, and the result is returned as a `tool` role message before the LLM synthesises the final answer.

The optimization target. KTCE and TroVE both maintain an evolving tool library, but their optimization strategies are orthogonal: KTCE applies an *explicit* LLM-driven delete/modify/add loop with loss-guided rollback, while TroVE applies *implicit* population pressure through frequency-based forgetting. CRAFT applies no post-creation optimization. SMITH's optimization is the RL training process itself; rather than refining individual tools after the fact, gradient descent directly improves the *policy* that generates tools, making each new tool better than the last.

The retrieval bottleneck. KTCE and CRAFT require a retrieval step before any tool can be used; retrieval quality therefore bounds solution quality. TroVE sidesteps this by injecting the entire library into the prompt, but this only works because frequency-based trimming keeps the library small enough to fit in context. SMITH eliminates retrieval entirely: the trained model writes the tool it needs from scratch in one pass, bypassing any index or library lookup.

L Baseline Failure Analysis

This appendix documents implementation-level failure modes we observed when evaluating KTCE [12] and TroVE [11] with Qwen3-4B-Instruct-2507 on our benchmark suite. These observations inform the token-cost and accuracy numbers reported in Table 1 and motivate several methodological choices in our evaluation pipeline.

L.1 KTCE: Three Compounding Failure Modes

Inspecting per-question inference outputs across all 24 tasks reveals three distinct failure modes that together explain KTCE's uneven task profile.

Failure Mode 1: Programmatic solver bypass. KTCE's inference code falls back to a hand-coded `solve` dispatcher when no LLM is configured or when the tool retrieval step returns an empty set. Tasks such as `bitwise_arithmetic`, `count_bits`, `tower_of_hanoi`, and `isomorphic_string` are solved entirely by this dispatcher, reaching 100% accuracy without a single LLM call. While this inflates overall averages, it also masks the failure of the tool-generation pipeline for those categories: the evolutionary toolset produced for these tasks was never invoked.

Conversely, tasks whose programmatic solver is incomplete or absent drop to near-zero: `cryptarithm` (5.7%), `knights_knaves` (4.8%), and `polynomial_equations` (1.1%) all fall into this category. `GQA` reaches 0% because the dispatcher has no implementation for visual question answering.

Failure Mode 2: Stub tool generation. For approximately half the tasks in our suite, KTCE's offline evolutionary loop produces a degenerate tool, specifically a function whose body is simply `return ""`. A representative example from the `ab` task:

```
def solve_ab(problem: str) -> str:
    return ""
```

When the LLM subsequently generates code that calls `solve_ab`, code execution succeeds but yields an empty string, causing the grader to mark the answer incorrect. Accuracy for stub-tool tasks therefore depends entirely on whether `_extract_final_answer` can recover a usable answer from the raw model response, effectively reducing KTCE to a plain chain-of-thought baseline. Tasks where the model can reason textually without the tool still score well (`syllogism`: 96.7%, `gcd`: 96.2%), while computation-heavy tasks that genuinely need a working tool collapse (`ab`: 0%, `group_anagrams`: 0%, `base_conversion`: 0%).

Failure Mode 3: LLM endpoint instability. Inspecting individual inference-output files reveals that many runs experienced intermittent LLM failures: the API call returned an empty response, causing `_solve_with_llm` to return early with no generated code and no token-usage record. For example, the `ab` task has only 64 of 210 entries with a real LLM response. This also corrupts the aggregated `all_results.json`, which can be silently overwritten by a subsequent failed re-run, replacing real API token counts with tokenizer-estimated zeros. We therefore read token usage from individual `inference_output/*.json` files and skip entries where `completion_tokens` = 0 when computing the Avg. Tokens figure in Table 1.

Per-task breakdown. Table 16 summarises all three failure modes across the full task suite.

L.2 TroVE: Missing Token-Usage Records

TroVE uses a two-phase pipeline: Phase 1 (validate split) grows the tool library; Phase 2 (test split, frozen library) is used for scoring. Token usage is stored in the results files only when the OpenAI-compatible backend records it. Inspecting the Phase 2 results across 20 tasks, 12 tasks have no stored `token_usage` whatsoever.

The `load_trove_token_average` function previously fell back to reconstructing the prompt by rendering TroVE's Mako templates with the *initial* three-function toolbox. However, Phase 2 prompts actually include the *learned* library from Phase 1, which can be substantially larger. Using the initial toolbox for reconstruction produces a prompt of only $\sim$179 characters ($\sim$45 tokens), far smaller than the real prompt, leading to a severely underestimated average of 691 tokens.

We correct this by computing the macro-average token count *only* over tasks that have real API-recorded usage (8 of 20 tasks), yielding 829 tokens. This figure is itself a lower bound: the 12 excluded tasks plausibly have larger prompts due to larger learned libraries, so the true TroVE average likely exceeds 829 tokens per question.

M TabMWP-Hard: Dataset Construction and Augmentation

Standard CoT achieves 96.8% on the original TabMWP benchmark (Table 1), making it an unreliable signal of tabular reasoning ability: a model that simply identifies the named entity in a small, clean table and reads off its value will score near-perfectly. This appendix documents the augmentation pipeline used to construct **TabMWP-Hard**, a harder evaluation set derived from the same problems, and illustrates each transformation step with concrete examples from the dataset.

M.1 Table Type Coverage

TabMWP tables are first assigned to one of seven structural types via a deterministic rule-based classifier (priority-ordered: STEM-AND-LEAF, FINANCIAL LEDGER, TWO-WAY, FUNCTION TABLE, PRICE RATE, PRICE LIST, NAMED COUNT). We select **three** types for full augmentation: FINANCIAL LEDGER, PRICE LIST, and TWO-WAY.

Table 16: Per-task KTCE diagnostic breakdown (Qwen3-4B-Instruct-2507). *Real LLM calls* counts entries with `completion_tokens` > 0 in the per-question inference output files. *Stub tool* indicates the generated tool body unconditionally returns an empty string.

Task	Acc. (%)	N	Real LLM calls	Stub tool	Failure mode
ab	0.0	210	64 / 210	Yes	FM2 + FM3
base_conversion	0.0	210	0 / 210	Yes	FM2 + FM3
bitwise_arithmetic	100.0	280	0 / 280	N/A	FM1 (solver)
caesar_cipher	82.4	210	0 / 210	No	FM1 (solver)
calendar_arithmetic	66.7	198	198 / 198	Yes	FM2
chinese_theorem	100.0	100	0 / 100	No	FM1 (solver)
complex_arithmetic	25.0	200	200 / 200	Yes	FM2
count_bits	100.0	210	0 / 210	No	FM1 (solver)
countdown	78.6	210	210 / 210	Yes	FM2
cryptarithm	5.7	210	0 / 210	No	FM1 (incomplete solver)
gcd	96.2	210	210 / 210	Yes	FM2
gqa	0.0	2516	0 / 2516	No	FM1 (no solver)
group_anagrams	0.0	200	200 / 200	Yes	FM2
gsm8k	50.9	1319	0 / 1319	No	FM1 (solver)
isomorphic_string	100.0	350	0 / 350	No	FM1 (solver)
knights_knaves	4.8	210	0 / 210	No	FM1 (incomplete solver)
lcm	18.1	210	210 / 210	Yes	FM2
polynomial_equations	1.1	280	0 / 280	No	FM1 (incomplete solver)
polynomial_mult.	66.1	280	0 / 280	No	FM1 (solver)
puzzle24	40.1	382	382 / 382	Yes	FM2
self_reference	75.6	234	234 / 234	Yes	FM2
simple_equations	16.0	16	16 / 16	Yes	FM2
syllogism	96.7	210	210 / 210	Yes	FM2
tabmwp	56.0	3152	0 / 3152	No	FM1 (solver)
tower_of_hanoi	100.0	210	0 / 210	No	FM1 (solver)

STEM-AND-LEAF tables are excluded because the key space is inherently bounded: stems are single digits 0–9, so at most $10 - |\text{existing stems}|$ distractor rows can ever be added. A table with five existing stems leaves room for at most five new ones, far too few to bury the target row or challenge a model's lookup ability. FUNCTION TABLE entries are similarly excluded because distractor rows must extend the exact linear sequence, leaving no freedom to generate confusingly adjacent keys. The three selected types admit arbitrarily large distractor pools and support both large-scale row injection and adversarial near-miss construction.

M.2 Augmentation Pipeline

Each selected entry passes through four sequential stages:

1. **Distractor row injection.** Up to 2,000 type-compatible rows are inserted at random positions, growing the table from a handful of rows to hundreds or thousands.
2. **Near-miss row insertion.** An LLM (or rule-based fallback) generates rows whose first-column key is *similar but distinct* from the key the question asks about, and places them immediately adjacent to the target row.
3. **LLM column augmentation.** A language model proposes 3–4 additional columns that are contextually plausible for the table's domain but do not help answer the question. For PRICE LIST tables, which carry no header in their original format, the LLM simultaneously names the existing columns and invents new ones, injecting a proper header row.
4. **Two-step validity gate.** The LLM first answers the question from the *original* table to confirm it can reach the correct answer, then verifies the augmented table still contains the necessary information. Entries that fail either step are discarded.

Entries that survive all four stages are further checked for key collisions (Section M.6) before being written to the final dataset.

M.3 Worked Example A: TWO-WAY Table

Original table. The original entry is a five-row, three-column philanthropic-donation table. The question asks for the difference between two specific people's donations to a specific cause.

Original table: TWO-WAY (5 rows × 3 columns)

Person	Animal rights	Clean water
Eve	$4	$15
Eli	$12	$5
Bridgette	$9	$11
Kamal	$18	$11
Janelle	$13	$13

Q: How much more money did Eve donate to clean water than Eli? **A:** $10

Answering requires only two lookups in a clean, five-row table, trivial for any language model that can parse basic tabular text.

After distractor injection. 2,000 rows are requested; the generator samples entity names from a dataset-wide pool, produces per-column values within each column's observed type and range (integer currency for this table), and shuffles all rows into random positions. The target rows (Eve and Eli) are buried at arbitrary offsets.

After near-miss insertion. The LLM identifies "Eve" and "Eli" as the two keys referenced by the question and inserts similarly-named rows immediately adjacent to each target. Because this is a person-name table, the generated near-miss keys are phonetically or orthographically close (e.g. "Ev", "Elliot") with deliberately different dollar values.

After LLM column augmentation. The LLM proposes four additional columns that fit a philanthropic context but do not encode the answer: *Membership Year*, *Donation Method*, *Annual Giving Level*, and *Recurring Donor*. Python snippets supplied by the LLM fill all rows with plausible values. The final table has **7 columns** and **308 rows**.

Augmented table: TWO-WAY (308 rows × 7 columns, excerpt)

Person	Animal rights	Clean water	Membership Year	Donation Method	Annual Giving Level	Recurring
...	...	...	...	...	...	...
Patty	$11	$6	2013	Check	Silver	F
Sports	$4	$13	2018	Bank Transfer	Platinum	T
Eve	$4	$15	2018	PayPal	Gold	T← target
Ev	$6	$9	2019	Bank Transfer	Silver	F← near-miss
Elliot	$10	$11	2021	Credit Card	Bronze	F← near-miss
Trisha	$7	$10	2015	Check	Platinum	T
...	...	...	...	...	...	...

A model reading this table must: (1) locate the correct row among hundreds, (2) ignore four irrelevant columns to read the right value, and (3) not be misled by the adjacent near-miss entries whose names differ by only one or two characters.

M.4 Worked Example B: PRICE LIST Table (Missing Header)

Original table. PRICE LIST tables in the original TabMWP dataset carry **no header row**: every line is a data row in the format `item name | $price`. The following entry has four rows and asks about the combined cost of two items.

Original table: PRICE LIST (4 rows, **no header**)

orange cone shell	$0.05
spiral snail shell	$0.03
purple clam shell	$0.03
scallop shell	$0.08

Q: Cassie has $0.18. How much money will Cassie have left if she buys a spiral snail shell and a scallop shell? **A:** $0.07

After distractor injection. New item–price rows are generated from a dataset-wide item pool, matched to the existing whole-cent price format and $0.03–$0.08 range. Items are paired to form compound names (e.g. `"bag of peanuts digital camera"`) so that distractor keys can never accidentally match a single-word original key.

After near-miss insertion. The LLM generates rows whose names are *semantically related but distinct* from the two answer items ("spiral snail shell" and "scallop shell"). Because size or variant suffixes (e.g. "spiral snail shell (small)") are explicitly forbidden in the prompt, the LLM instead invents natural-language variants: *spiral whelk shell*, *striped snail shell*, *scallop valve*, *giant scallop shell*. These are inserted near the target rows so that a model skimming for "shell" will encounter multiple plausible but incorrect candidates.

After LLM column augmentation (with header injection). Because the original table has no header, the LLM is asked to perform two tasks simultaneously: (a) name the two existing columns, and (b) propose 3–4 new columns. It returns names for the existing columns (*Shell Type*, *Unit Price*) plus new columns (*Inventory Count*, *Origin Region*, *Shell Grade*, *Supplier ID*). A header row is constructed and prepended to the table. The final table has **6 columns** and **316 rows**.

Augmented table: PRICE LIST (316 rows × 6 columns, excerpt)

Shell Type	Unit Price	Inventory Count	Region	Grade	Supplier ID	
...	...	...	...	...	...	
bag of peanuts ...	$0.06	126	Arctic	C	SUP-1083	
scallop shell	$0.08	405	Atlantic	C	SUP-1084	← target
night's stay at ...	$0.05	158	Pacific	B	SUP-1085	
...	...	...	...	...	...	
spiral snail shell	$0.03	400	Arctic	C	SUP-1232	← target
spiral whelk shell	$0.06	10	Atlantic	B	SUP-1240	← near-miss
striped snail shell	$0.04	316	Arctic	C	SUP-1241	← near-miss
scallop valve	$0.07	175	Indian	C	SUP-1242	← near-miss
giant scallop shell	$0.12	260	Pacific	C	SUP-1243	← near-miss
...	...	...	...	...	...	

This example illustrates two difficulties simultaneously: the missing-header case that forces the LLM to infer column semantics before it can construct the header, and the cluster of near-miss shell names with plausible but wrong prices that surrounds both target rows.

M.5 LLM Column Proposal Mechanism

Column augmentation is performed by `gpt-oss-120b` via the Together API. To keep the prompt tractable regardless of how many distractor rows were added, the LLM is shown only the *original* (unaugmented) table for domain context; the number of rows to fill, n, is derived from the *full* augmented table.

For each proposed column, the LLM supplies a short Python snippet that builds a list called `values` with exactly n elements. The snippet may use `random`, `math`, and `string` from the standard library, enabling varied but deterministic outputs (seeded per entry). A hard deduplication pass discards any proposed column whose name (case-insensitive) already exists in the table, preventing the LLM from reinstating existing columns under a slightly different spelling. At most four new columns are accepted per entry.

For PRICE LIST tables the prompt is extended to ask for names for the existing columns as well, using the format:

```
{"existing_column_names": ["Shell Type", "Unit Price"],
"columns": [{"column_name": "Inventory Count",
"python_code": "values = [random.randint(1,500) for _ in
range(n_rows)]"},
...]}
```

The injected header is then built by concatenating the existing-column names and the new-column names in order, ensuring a well-formed header even for tables that were originally headerless.

M.6 Collision Avoidance

Two independent mechanisms ensure that no distractor or near-miss row introduces an ambiguous or incorrect ground truth.

Key-level collision guard. The distractor generators maintain the set of first-column keys present in the original table and filter every candidate key against it before inclusion. For PRICE LIST tables, compound item names (e.g. "apple bread" from pairing "apple" and "bread" from the pool) prevent accidental single-word matches. For FINANCIAL LEDGER tables, new transactions reuse only dates already present in the original ledger; extending the date range would shift "end-of-period" balance answers. After all rows are merged, a deterministic validator scans the augmented table and flags any original key that appears more than once; entries with any such collision are discarded.

Answer-value near-miss validator. Near-miss rows are designed to confuse the model's *row selection*, not to carry a numerically correct value. After near-miss insertion, the LLM is given the list of newly added rows, the question, and the ground-truth answer, and asked to issue a KEEP or REMOVE decision for each row according to four criteria: (i) the row's values would accidentally reproduce the correct answer when used with the original table; (ii) the row's first-column key already exists in the original table (duplicate key); (iii) the row's value format is inconsistent with the table (e.g. missing $ signs in a currency table); (iv) the row's label comes from a different semantic domain than the other row labels (e.g. a sport name in a table of person names). A rule-based pre-filter handles criterion (ii) before the LLM call, removing exact-key duplicates without spending API tokens. Rows flagged REMOVE are deleted from the augmented table; entries where no near-miss rows survive the filter are discarded entirely rather than retained without adversarial pressure.

N Compute and Training Cost

All SMITH training runs were performed on a single NVIDIA RTX 6000 Pro GPU. Each training session, covering one model size and one tool-pool configuration, took between 36 and 72 hours to complete depending on rollout length, number of pooled tools, and the size of the policy backbone (Qwen3-4B, Qwen3-8B, or Granite-3.3-8B). We report this to make the cost of reproducing our results explicit: the recipe is reachable on a single workstation-class GPU and does not require multi-node clusters.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction explicitly state all core claims: (1) joint RL training of tool creation and use inside a single policy, (2) three disentangled reward axes (execution, judge, format), (3) 79.8 macro-average on held-out Reasoning-Gym tasks, (4) zero-shot transfer to TabMWP-Hard (40.4) and GQA (42.6), and (5) scalability to Qwen3-8B and Granite-3.3-8B. Each claim is directly supported by Tables 1–4 and the ablation study.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: A dedicated Limitations section discusses: (1) scope, SMITH covers only procedural/computation tasks with exact verifiers from Reasoning-Gym and does not address open-ended retrieval, web navigation, or embodied control; (2) tool design, tools are stateless Python functions paired with JSON schemas, used in isolation for at most 5 turns with one domain tool and two distractors per prompt, leaving multi-tool composition, longer-horizon workflows, and richer protocols (MCP, Agent Skills) out of scope; (3) scale and judge dependence, policies are at most 8B parameters and the quality judge has 30B activated parameters, with open questions at 70B+ and the Self-Judge ablation only partially probing judge-size dependence; and (4) safety and evaluation gaps, generated Python is sandboxed at every step but not formally certified against adversarial prompts inducing unsafe tool execution, and we report no human evaluation of schema readability or developer-facing reusability.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: The paper makes no formal theoretical claims and contains no theorems, lemmas, or proofs. All contributions are algorithmic (reward design, training framework) and empirical.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Section 3 and Section 4 together specify the model (Qwen3-4B-Instruct), fine-tuning method (LoRA, r=64, α=128), optimizer (DAPO), key hyperparameters (β=0.01, lr 6×10^{-5}, temperature 0.7, 60 gradient steps, n_{gen}=8 rollouts), reward formulas (Eqs. 1–4), the 13 training task categories, the easy-to-hard split protocol, and the evaluation benchmarks. All prompt templates and judge prompts are reproduced verbatim in the appendix.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification: Code is not released at submission time to preserve anonymity. Training data is generated on-the-fly from Reasoning-Gym [18], which is publicly available. TabMWP-Hard, the new evaluation dataset, is described in Appendix M with full construction details. We plan to release code and TabMWP-Hard upon acceptance.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.

- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 4 (Experiments) specifies the 13 training task categories, 10 held-out evaluation tasks, out-of-domain benchmarks (TabMWP-Hard, GQA), model and LoRA configuration, DAPO RL training settings (KL coefficient, learning rates, rollouts per prompt, temperature, gradient steps), reward coefficients, tool pool capacity, and the easy-to-hard difficulty split protocol. Per-task difficulty splits for all 13 categories are listed in Appendix F. Baseline reimplementation details and inference protocols are provided in the main text and appendices.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: Results are reported as single-run accuracy figures without error bars or confidence intervals. This is due to the high computational cost of RL training runs at 4B and 8B scale. The main comparison (Table 1) uses a fixed test set of at least 100 instances per category, which partially mitigates variance at the macro-average level, but we acknowledge the absence of multi-seed statistics as a limitation.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
   - If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Appendix N reports that all SMITH training runs were performed on a single NVIDIA RTX 6000 Pro GPU, with each run taking 36–72 hours depending on model size (Qwen3-4B, Qwen3-8B, Granite-3.3-8B), rollout length, and tool-pool configuration.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.

- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research develops a training framework for LLM tool creation and does not involve human subjects, sensitive personal data, or dual-use weapons research. We have reviewed the NeurIPS Code of Ethics and confirm compliance. Generated Python tools are executed in an isolated sandbox environment.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: A dedicated Broader Impact section (Appendix A) discusses both sides. Positive impacts: smaller open-weight models become more competitive at agentic reasoning, reducing dependence on closed frontier APIs; jointly training the writer and user surfaces failures during training rather than at deployment. Negative impacts: autonomous code synthesis raises sandboxing concerns; the paper recommends restricted Python sandboxes with no network access and notes the tools target procedural reasoning rather than high-risk capabilities.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [Yes]

    Justification: The Limitations section explicitly states that generated Python tools are executed inside a sandbox at every training step, and recommends that downstream deployments retain sandboxing guarantees and add content-level filtering before invoking generated tools on untrusted inputs. Code and model weights are not released at submission time.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.

- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All assets are cited: Reasoning-Gym [18] (Apache 2.0), TabMWP [29] (MIT), GQA [30] (CC BY 4.0), Qwen3 [32] (Apache 2.0), Granite-3.3 (Apache 2.0), LFM2.5 [33], and all baseline methods (LATM, CRAFT, Trove, KTCE, ReTool) are cited. Explicit license names are not stated inline in the paper text; we will add a license table to the appendix.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The paper introduces TabMWP-Hard, a harder variant of TabMWP with tables scaled to up to 5,000 rows and added distracting columns. Its construction procedure, motivation, and statistics are documented in Appendix M. The fine-tuned model checkpoints (SMITH Qwen3-4B, 8B, Granite-3.3-8B) are not released at submission time to preserve anonymity; they will be released upon acceptance.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve any crowdsourcing, human annotation, or research with human subjects. All training and evaluation data are either procedurally generated (Reasoning-Gym, TabMWP-Hard) or drawn from existing benchmarks (TabMWP, GQA).

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: No human subjects are involved in this research. IRB approval is not required.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [N/A]

Justification: LLMs were not used as a writing or editing tool in preparing this manuscript in any way that impacts scientific rigor or originality. The LLMs appearing in the paper (Qwen3, Granite, LFM2.5, Qwen3-30B-A3B as judge) are the research subjects themselves and are fully described in Section 3; they do not constitute a non-standard editorial usage requiring declaration under this item.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.