GEAR: Training-Free Rule Distillation for Advanced and Efficient Tool-Augmented Reasoning

Bhiman Kumar Baghel[1,2]* **Fanyou Wu**[2] **Suraj Maharjan**[2]
Peiran Yao[2] **Srinivasan Sengamedu**[2]
[1]University of Pittsburgh [2]Amazon
Correspondence: bkb45@pitt.edu

Abstract

Improving tool use in large reasoning models (LRMs) typically demands expensive test-time compute, training resources unavailable to API-only practitioners, or expensive iterative rollouts with execution feedback required by training-free context-engineering methods, all incompatible with resource-constrained deployment. We introduce **GEAR**, a training-free framework that extracts compositional rules from tool-use failures via single-pass meta-reasoning and distills them into inference prompts. We hypothesize that combining explicit rules with examples outperforms either alone. Across 78 API domains, **GEAR**+2-shot outperforms 3-shot by 24.9% on Qwen3-30B, validating our complementarity hypothesis. On Qwen3-4B, **GEAR** exceeds LoRA SFT by 24.3% and RAG-3-shot by 38.5%, while **GEAR**+2-shot beats the state-of-the-art context-engineering method ACE by 38.8% at 4.1× lower cost. Remarkably, Qwen3-30B with **GEAR**+2-shot nearly doubles zero-shot accuracy while remaining under 1 s latency, demonstrating rollout-free adaptation for resource-constrained deployment.

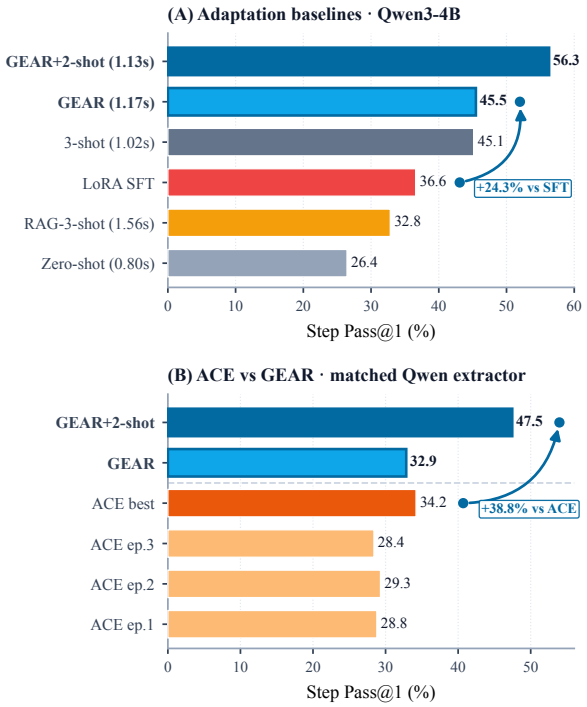


Figure 1: On Qwen3-4B, **GEAR** outperforms LoRA SFT (+24.3%) and RAG-3-shot (+38.5%); **GEAR**+2-shot nearly doubles zero-shot Step Pass@1 (A). With a matched Qwen extractor, **GEAR**+2-shot exceeds ACE's best playbook by +38.8% (B).

1 Introduction

Tool use is essential for AI agents to interact with external systems for real-world tasks. Large reasoning models (LRMs) excel at familiar API domains but struggle to adapt to specialized domains underrepresented during pretraining. Test-time compute techniques such as self-reflection (Yao et al., 2023b) and multi-path exploration (Yao et al., 2023a) address this but substantially increase latency and cost. Distillation (Schick et al., 2023; Qin et al., 2024; Patil et al., 2024; Chen et al., 2024; Ning et al., 2024) and fine-tuning of resource-intensive open-source (Guo et al., 2025; Yang et al., 2025; Team et al., 2025) LRMs require training resources unavailable to most practitioners. Recent training-free context-engineering methods such as ACE (Zhang et al., 2026) and Training-Free GRPO (Cai et al., 2025) sidestep training but require expensive iterative rollouts with execution feedback. We target training-free tool-use adaptation for resource-constrained deployment: no model training, no online rollouts, and limited per-domain data.

In-context learning (ICL) offers a training-free path: few-shot prompting (Brown et al., 2020) employs inductive reasoning over task-specific demonstrations but is limited by combinatorially large failure spaces and context window constraints.

*Work done during internship at Amazon.

Proceedings of the 15th International Joint Conference on Natural Language Processing and the 5th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics, pages 4781–4820
November 6-10, 2026 ©2026 Association for Computational Linguistics

Similarity-based retrieval (Zhang et al., 2025) addresses coverage but adds latency and, as we show, can underperform random selection. We argue tool-use failures exhibit recurring patterns capturable as explicit rules, providing deductive guidance complementary to inductive examples. We hypothesize that combining the two outperforms either alone.

We introduce **GEAR** (**G**ather, **E**xtract, Rule-**A**ugmented **R**easoning), a training-free framework that operationalizes this hypothesis through three stages: (1) *gather* baseline predictions capturing both successes and failures, (2) *extract* compositional rules through meta-reasoning over failure patterns, and (3) *provide* rule-augmented reasoning by distilling rules into inference prompts. Central to **GEAR** is a domain-agnostic meta-prompt, optimized once through LRM self-critique, that produces domain-specific rules in a single inference call per domain. Unlike APO methods and their recent context-engineering variants (Zhou et al., 2023; Yang et al., 2024; Zhang et al., 2026; Cai et al., 2025), **GEAR** requires no rollouts or performance feedback at any stage, achieving 4.1× lower cost than ACE (Section 6).

Across 78 API domains, **GEAR**+2-shot achieves the highest accuracy in every model and configuration tested, validating our complementarity hypothesis. On Qwen3-4B, **GEAR** outperforms LoRA SFT and RAG retrieval, while **GEAR**+2-shot exceeds ACE by 38.8% at 4.1× lower cost and nearly doubles zero-shot Step Pass@1 (Figure 1). Remarkably, the same **GEAR**+2-shot recipe on Qwen3-30B nearly doubles zero-shot accuracy while remaining under 1 s latency, demonstrating rollout-free tool-use adaptation for resource-constrained deployment.

2 Related Work

Training-based approaches such as ToolFormer (Schick et al., 2023), ToolLLM (Qin et al., 2024), Gorilla (Patil et al., 2024), and others (Li et al., 2023; Chen et al., 2024; Ning et al., 2024) fine-tune models or distill LRM capabilities into smaller models, requiring model weights and training compute unavailable to most practitioners. **GEAR** operates only through prompting.

Test-time reasoning methods such as self-reflection (Yao et al., 2023b), multi-path exploration (Yao et al., 2023a), chain-of-thought (Wei et al., 2022), and self-consistency (Wang et al., 2023) improve accuracy but increase latency. Per-instance reflection methods (Shinn et al., 2023; Madaan et al., 2023; Bai et al., 2022) require trajectory-specific replay. **GEAR** performs cross-instance learning, extracting reusable rules from historical failures that transfer without replay.

To enhance ICL, prior work (Zheng et al., 2025; Stengel-Eskin et al., 2024; Wang et al., 2024; Murty et al., 2024; Zhang et al., 2025; Qian et al., 2023; Shang et al., 2025) acquires reusable code, skills, and APIs through exploration, with retrieval (Lewis et al., 2020; Zhang et al., 2025) selecting relevant components per query at additional latency cost. **GEAR** instead extracts compact behavioral rules from failures, statically included in prompts without retrieval, complementing few-shot ICL by preventing errors rather than expanding capabilities.

Automated prompt optimization (APO) refines instructions and demonstrations from data. Classical methods (Zhou et al., 2023; Yang et al., 2024; Fernando et al., 2024; Opsahl-Ong et al., 2024) use meta-prompting, evolutionary search, or reflective evolution; GEPA (Agrawal et al., 2025) learns high-level rules from trajectories. Recent context-engineering variants ACE (Zhang et al., 2026) and Training-Free GRPO (Cai et al., 2025) extend this through iterative generate-reflect-curate cycles or group-relative advantages over rollouts, embedding knowledge into monolithic prompts that saturate over iterations. **GEAR** differs: it extracts interpretable, compositional rules complementing few-shot examples, with no performance feedback during meta-prompt optimization (run once across all domains) or rule extraction (single-pass, rollout-free). Empirically, **GEAR** outperforms ACE by 38.8% step accuracy at 4.1× lower cost (Section 6).

3 Problem Formulation

We formalize tool-augmented reasoning as a sequential decision-making task where the agent orchestrates external API calls to answer user queries.

3.1 Tool-Use as a Markov Decision Process

We model tool-use as an episodic Markov Decision Process (MDP), comprising state space $\mathcal{S}$, action space $\mathcal{A}$, transition function $\mathcal{P}$, and reward function r. At each timestep t, the agent observes state $s_t = (\mathcal{I}, \mathcal{D}_{\text{API}}, q, \mathcal{H}_t)$, comprising system instructions $\mathcal{I}$, API documentation $\mathcal{D}_{\text{API}}$, user query q, and trajectory history $\mathcal{H}_t = \langle(\alpha_0, o_0), \ldots, (\alpha_{t-1}, o_{t-1})\rangle$ of previous action-observation pairs. The agent

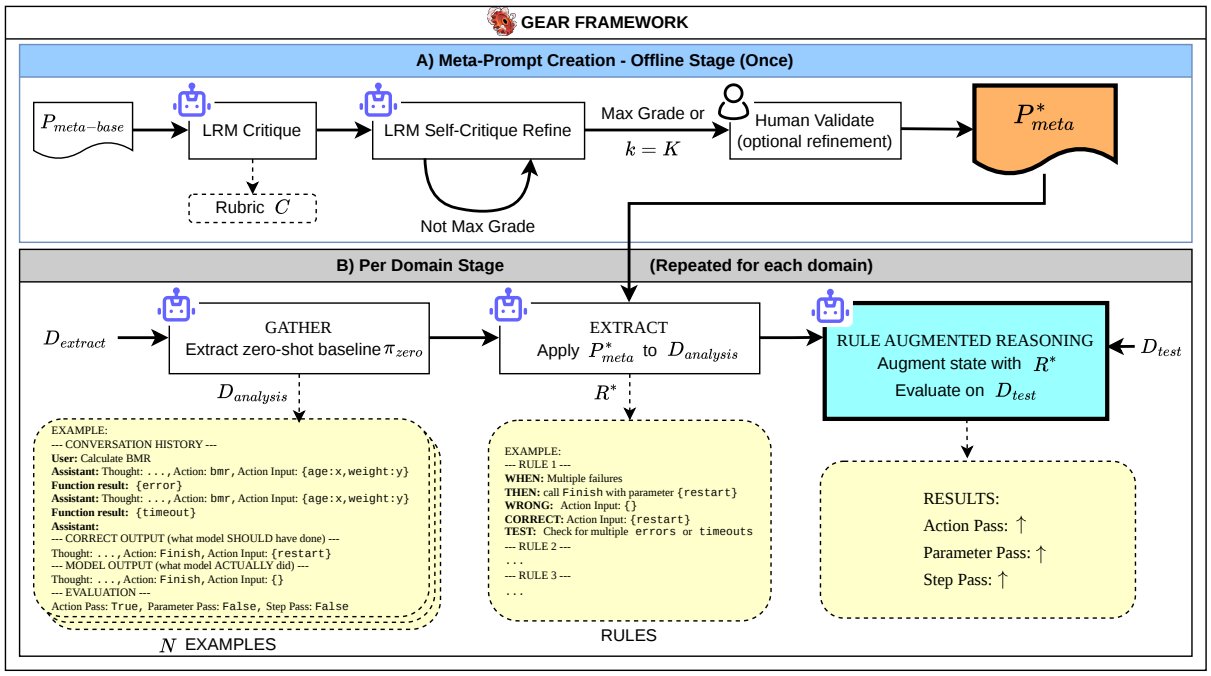


Figure 2: **GEAR** framework overview. The **offline stage** creates a domain-agnostic meta-prompt $\mathcal{P}^*_{\text{meta}}$ through iterative LRM self-critique with optional human validation. The **per-domain stage** executes three phases: Gather (collect baseline predictions), Extract (synthesize rules via meta-reasoning), and Rule-Augmented Reasoning (evaluate with rule-augmented state). The meta-prompt is created once and reused across all domains.

generates an action $\alpha_t = (a_t, \theta_t)$ consisting of an API identifier a_t and corresponding parameters θ_t, following a policy $\pi : \mathcal{S} \to \mathcal{A}$ that maps states to actions. This process continues until a termination action provides the final answer.

While the deployment goal is producing correct final answers through full trajectory execution, this work focuses on **per-step action prediction accuracy**: given a state s_t with ground-truth action α^*_t, we seek policies that maximize $\mathbb{P}[\pi(s_t) = \alpha^*_t]$. This step-wise paradigm, motivated by Chen et al. (2024)'s finding that step-wise supervision outperforms full-trajectory alternatives for LLM tool-use reasoning, provides fine-grained feedback on realized execution traces. LLM-as-judge filtering (Appendix F) ensures each gold action is a validated optimal continuation rather than an arbitrary single path, and evaluation instances span trajectory histories $\mathcal{H}_t$ of varying hop lengths. See Appendix B for complete formalization.

3.2 Rule-Augmented Formulation

Contemporary approaches implement the policy π through in-context learning (ICL), employing inductive reasoning over demonstration examples to generalize to new queries. We hypothesize that combining inductive learning with deductive reasoning over explicit constraints provides richer guidance than either alone. While few-shot learning requires models to infer patterns from examples, explicit rules enable direct constraint application. We formalize baseline approaches in Appendix C.

GEAR augments the state with explicit domain-specific behavioral rules $\mathcal{R} = \{r_1, r_2, \ldots, r_{|\mathcal{R}|}\}$, yielding $s_t = (\mathcal{I}, \mathcal{D}_{\text{API}}, \mathcal{R}, q, \mathcal{H}_t)$. These rules remain static across all queries within the same domain. Each rule $r_i = (\text{Directive}_i, \text{Pedagogical}_i)$ comprises two components: the directive specifies when the rule applies and the required behavior (WHEN-THEN structure), while the pedagogical component illustrates correct and incorrect behaviors along with test criteria for verification. This structure enables deductive constraint application through both explicit directives and concrete behavioral guidance. **GEAR** is complementary to few-shot learning: combining rules with demonstration examples $\mathcal{E}$ yields $s_t = (\mathcal{I}, \mathcal{D}_{\text{API}}, \mathcal{R}, \mathcal{E}, q, \mathcal{H}_t)$, leveraging both deductive and inductive reasoning as validated empirically in Section 6.

3.3 Rule Extraction Problem

The rule extraction problem seeks a compact rule set $\mathcal{R}^*$ that maximizes per-step accuracy on a held-out test set $\mathcal{D}_{\text{test}}$: $\mathcal{R}^* = \arg\max_{\mathcal{R}} \text{Accuracy}(\pi_{\mathcal{R}} \mid \mathcal{D}_{\text{test}})$ subject to three constraints: interpretability (natural language rules), generalization (class-level patterns rather than instance memorization), and compactness (minimal rule count). This is a discrete combinatorial optimization over natural language rule sets, precluding gradient-based methods. Section 4 describes our single-pass meta-reasoning approach to this problem.

3.4 Meta-Prompt Creation Problem

The quality of extracted rules depends critically on the meta-prompt $\mathcal{P}_{\text{meta}}$ used for rule extraction. We formulate meta-prompt optimization as finding an optimal meta-prompt $\mathcal{P}^*_{\text{meta}}$ that maximizes rule effectiveness: $\mathcal{P}^*_{\text{meta}} = \arg\max_{\mathcal{P}} \mathbb{E}_{\mathcal{R}\sim\text{Extract}(\mathcal{P},\mathcal{D}_{\text{analysis}})}[\text{Accuracy}(\pi_{\mathcal{R}} \mid \mathcal{D}_{\text{test}})]$ where $\mathcal{D}_{\text{analysis}}$ contains baseline predictions for failure analysis. Evaluating this objective requires expensive rule extraction and inference for each candidate prompt, making exhaustive search intractable. Section 4.1 describes our efficient LRM-based self-critique approach that optimizes without external feedback.

4 GEAR Framework

GEAR operates through two stages (Figure 2). An offline stage creates a domain-agnostic meta-prompt $\mathcal{P}^*_{\text{meta}}$ via iterative self-critique (Section 4.1), reused across all domains without modification. A per-domain stage then applies the fixed $\mathcal{P}^*_{\text{meta}}$ through three phases: Gather, Extract, and Rule-Augmented Reasoning (Sections 4.2–4.4).

4.1 Meta-Prompt Creation

Rule extraction quality depends critically on the meta-prompt design. Traditional APO approaches, including recent context-engineering methods, evaluate candidates on train data or rollouts, requiring expensive inference per iteration. **GEAR** instead uses LRM-based self-critique for efficient refinement without external feedback.

Algorithm 1 (Appendix D) details the two-stage process. In Stage 1, an LRM generates an evaluation rubric $\mathcal{C}$ and critiques a base meta-prompt $\mathcal{P}_{\text{meta-base}}$. In Stage 2, the LRM iteratively refines this prompt using self-critique against the fixed rubric $\mathcal{C}$ until reaching the maximum grade or iteration limit K. Each iteration requires only two LRM calls (refine and self-critique) with no rollouts or test-set evaluation, the steps that make APO candidate evaluation expensive. An optional human sanity check verifies structural correctness (input placeholders, output schema) of the resulting domain-agnostic $\mathcal{P}^*_{\text{meta}}$, since LRMs can be overconfident in self-assessment. Optimization converged in five iterations (D- $\rightarrow$ A+ rubric grade); the trajectory, intermediate prompts, and rule-format validation are in Appendix D.

Algorithm 1 Meta-Prompt Creation

Input: Base meta-prompt $\mathcal{P}_{\text{meta-base}}$, iteration limit K
Output: Optimized meta-prompt $\mathcal{P}^*_{\text{meta}}$
// Stage 1: Rubric Generation and Initial Critique
$(\mathcal{C}, \{g_i^{(0)}\}) \leftarrow \text{LRM-Critique}(\mathcal{P}_{\text{meta-base}})$
$c_0 \leftarrow (\mathcal{C}, \{g_i^{(0)}\})$; $\mathcal{P}_0 \leftarrow \mathcal{P}_{\text{meta-base}}$
// Stage 2: Iterative Refinement
for $k = 1$ to K **do**
  $\mathcal{P}_k \leftarrow \text{LRM-Refine}(\mathcal{P}_{k-1}, c_{k-1})$
  $\{g_i^{(k)}\} \leftarrow \text{LRM-SelfCritique}(\mathcal{P}_k, \mathcal{C})$
  $c_k \leftarrow (\mathcal{C}, \{g_i^{(k)}\})$
  if $\text{MaxGrade}(\{g_i^{(k)}\})$ **then**
    break
  end if
end for
// Human Validation (optional)
$\mathcal{P}^*_{\text{meta}} \leftarrow \text{HumanValidate}(\mathcal{P}_K)$
return $\mathcal{P}^*_{\text{meta}}$

4.2 Phase 1: Gather

We partition data into extraction set $\mathcal{D}_{\text{extract}}$ and held-out test set $\mathcal{D}_{\text{test}}$, then execute a zero-shot policy π_{zero} on $\mathcal{D}_{\text{extract}}$ to generate the analysis dataset $\mathcal{D}_{\text{analysis}} = \{(s_t^{(i)}, \alpha_{\text{pred}}^{(i)}, \alpha_*^{(i)}, e^{(i)})\}_{i=1}^N$, where $\alpha_{\text{pred}}^{(i)}$ and $\alpha_*^{(i)}$ are the predicted and ground-truth actions and $e^{(i)}$ is a binary correctness indicator. Importantly, $\mathcal{D}_{\text{analysis}}$ includes both successes and failures, enabling contrastive analysis that prevents regression in Extract (Section 8).

4.3 Phase 2: Extract

The Extract phase applies the domain-agnostic meta-prompt $\mathcal{P}^*_{\text{meta}}$ to $\mathcal{D}_{\text{analysis}}$ in a single inference call per domain to synthesize domain-specific rules. The meta-prompt guides the LRM to: (1) categorize failures by type, (2) identify recurring patterns across multiple examples (frequency threshold $\geq \tau$, preventing instance memorization), (3) determine root causes through comparative analysis of correct versus incorrect predictions, and (4) synthesize rules following the two-tuple structure $r_i = (\text{Directive}_i, \text{Pedagogical}_i)$ from Section 3.2.

The extraction produces $\mathcal{R}^* = \{r_1, \ldots, r_{|\mathcal{R}^*|}\}$, with $|\mathcal{R}^*|$ varying by per-domain failure diversity. Both the analysis process and rule schema are encoded within $\mathcal{P}^*_{\text{meta}}$, requiring no per-domain customization. Example rules and the complete schema appear in Appendix E.

4.4 Phase 3: Rule-Augmented Reasoning

This phase augments the agent's state to $s_t = (\mathcal{I}, \mathcal{D}_{\text{API}}, \mathcal{R}^*, q, \mathcal{H}_t)$ (Section 3.2). Rules guide both action selection and parameter population, requiring no additional computation beyond the standard forward pass. Rules are modular: they combine with few-shot examples or deploy compositionally via priority-ranked subsets, with priorities assigned during extraction by estimated error coverage.

5 Experimental Setup

5.1 Dataset

We use ToolPreference (Chen et al., 2024), which augments ToolBench (Qin et al., 2024) trajectories with action-level preference pairs. To our knowledge, it is the only tool-use dataset providing the action-level signals required for per-step evaluation; alternatives like Toucan (Xu et al., 2025) lack such supervision. The dataset also includes failed API calls with recovery steps reflecting real-world errors, and 78 diverse API domains spanning specialized tool ecosystems. We apply LLM-as-judge filtering to remove noisy pairs where neither action is optimal (Appendix F). After filtering, we retain 1,949 examples split 70:30 into $\mathcal{D}_{\text{extract}}$ (n=1,397) and $\mathcal{D}_{\text{test}}$ (n=552). Statistics in Appendix G.

5.2 Models and Configuration

We evaluate **GEAR** on three open-source models (Qwen3-4B, Qwen3-30B, Ministral3-14B) with greedy decoding (T=0), matching our efficiency focus. Model details in Appendix H.

Rule extraction uses Qwen3-30B with greedy decoding (T=0) and frequency threshold τ=3, balancing pattern reliability against coverage given limited per-domain examples ($\sim$18 on average), producing 6–12 rules per domain. We run extraction three times to mitigate non-determinism and select the rule set with best $\mathcal{D}_{\text{extract}}$ performance.

5.3 Baselines

Prompts follow the ReAct format (Yao et al., 2023b), with Thought-Action-Observation history $\mathcal{H}_t$. We compare **GEAR** against five baselines: (1) **Zero-shot** (standard ReAct, no examples or rules); (2) **Zero-shot-CoT** (with explicit step-by-step reasoning); and (3–5) **1/2/3-shot** (random examples from $\mathcal{D}_{\text{extract}}$ to measure example-scaling).[1] Sampling-based and reflection methods are excluded as latency-prohibitive (orthogonal to our efficiency focus). Prompts in Appendix I.

We additionally compare against three stronger adaptation baselines on Qwen3-4B, the scale where these methods are realistic: (i) **LoRA SFT**, fine-tuning on $\mathcal{D}_{\text{extract}}$; (ii) **RAG-3-shot**, retrieving the three most similar demonstrations from $\mathcal{D}_{\text{extract}}$ via sentence embeddings, in two variants keyed on user query ($\mathcal{R}_q$) or query plus trajectory ($\mathcal{R}_{\text{full}}$), also run on Qwen3-30B for cross-scale confirmation; and (iii) **ACE** (Zhang et al., 2026), the state-of-the-art context-engineering method (shown to outperform GEPA (Agrawal et al., 2025)), in offline mode. LoRA is infeasible for API-only or large MoE models, and ACE's rollout-driven cost is prohibitive at larger scales. Implementation details, hyperparameters, and two correctness fixes to ACE are in Appendix J.

5.4 GEAR Variants

Beyond core **GEAR**, we evaluate two variants testing deployment flexibility.

Complementarity. Rules enable deductive constraint application; examples enable inductive pattern inference. We hypothesize these complementary mechanisms address distinct failures or reinforce shared ones, and test this via **GEAR**+2-shot, combining rules with few-shot examples. We further test ordering: **GEAR**+2-shot (rules first) versus 2-shot+**GEAR** (examples first). We hypothesize that placing rules first establishes constraints framing the action space, with examples demonstrating how to satisfy them ("constrain-then-demonstrate").

Compositionality. For efficiency-sensitive applications, rules can be deployed compositionally. During extraction, rules are categorized into priority buckets (critical, high, medium, low) by estimated error coverage. We evaluate Critical-**GEAR**, deploying only critical-priority rules, hypothesizing it reduces latency while retaining most of **GEAR**'s effectiveness since high-priority rules

[1] We limit to 3-shot as 55% of domains contain only 6 test examples, risking demonstration-test overlap.

Method	Pass@1 (%)			Gain (%)	Efficiency			
	Action	Parameter	Step	ΔStep (%)	Latency (s)	ΔLat. (%)	In Tok.	Out Tok.
Qwen3-30B-A3B								
Zero-shot	57.53	42.72	30.20	–	**0.63**	–	1,299	83
Zero-shot-CoT	57.94	41.35	30.68	+1.6	0.64	−2.7	1,313	89
1-shot	64.64	51.36	38.84	+28.6	0.68	−7.8	1,823	89
2-shot	66.43	54.05	43.11	+42.7	0.69	−9.9	2,366	89
3-shot	70.97	58.00	47.78	+58.2	0.69	−9.6	2,961	90
GEAR (ours)	72.81	56.25	47.74	+58.1	0.85	−36.4	2,850	114
Critical-**GEAR** (ours)	65.25	49.91	38.40	+27.2	0.72	−14.2	1,858	101
GEAR+2-shot (ours)	**82.25**⋆	**65.24**⋆	**59.69**⋆	+97.6	0.78	−24.4	4,446	101
2-shot+**GEAR** (ours)	78.68	63.85	57.79	+91.4	0.80	−27.6	4,469	106
DirectiveOnly-**GEAR**	65.00	51.38	40.97	+35.7	0.83	−31.7	2,104	116
PedagogyOnly-**GEAR**	66.35	51.40	41.61	+37.8	0.75	−19.3	2,070	105
Qwen3-4B								
Zero-shot	57.53	37.57	26.45	–	**0.80**	–	1,299	84
Zero-shot-CoT	57.57	39.28	28.07	+6.1	0.86	−7.2	1,313	92
1-shot	62.88	48.93	37.78	+42.8	0.89	−11.4	1,823	93
2-shot	63.72	51.21	39.79	+50.4	0.94	−17.5	2,366	97
3-shot	68.09	57.41	45.15	+70.7	1.02	−27.0	2,961	104
GEAR (ours)	72.51	53.55	45.48	+71.9	1.17	−46.1	2,854	121
Critical-**GEAR** (ours)	63.42	45.17	33.96	+28.4	0.99	−24.4	1,830	104
GEAR+2-shot (ours)	**76.72**	**62.97**	**56.29**	+112.8	1.13	−41.4	4,268	108
2-shot+**GEAR** (ours)	72.22	58.07	47.48	+79.5	1.32	−64.5	3,917	132

Table 1: **GEAR** results on $\mathcal{D}_{\text{test}}$. Step Pass@1 (%) is the primary metric. ΔStep / ΔLat. = relative change vs each model's own Zero-shot (positive ΔLat. = faster). Best results per model in **bold**; ⋆overall best across configurations. Row colors: **GEAR** variants; component ablations. Full results in Appendix K.

capture the majority of the error distribution; results in Table 1.

5.5 Evaluation Metrics

We use three Pass@1 metrics: **Action** (correct API), **Parameter** (exact match of all parameter names and values, order-independent), and **Step Accuracy** (complete action correctness, our primary metric).[2] We also report **latency** (seconds from input to final token). All numbers are single-run means across 78 domains in $\mathcal{D}_{\text{test}}$.

6 Results

Table 1 reports results on Qwen3-30B and Qwen3-4B; Ministral3-14B and extended ablations appear in Appendix K. Adaptation baselines on Qwen3-4B appear in Tables 2 and 3.

Qwen3-30B with **GEAR**+2-shot achieves 59.69% Step Accuracy, the highest across all configurations in Table 1, nearly doubling zero-shot performance (+97.6%) while remaining under 1 s latency (0.78s). On the same model, **GEAR** alone already matches 3-shot prompting (47.74% vs 47.78%), suggesting that explicit rules can substitute for additional demonstrations when model capacity is limited. Qwen3-4B shows the same pattern at smaller scale: **GEAR**+2-shot reaches 56.29% Step Pass@1 (+112.8% over zero-shot), outperforming both **GEAR** alone and all few-shot baselines.

Complementarity. **GEAR**+2-shot exceeds both **GEAR** alone (+25.0%) and 2-shot alone (+38.5%) on Qwen3-30B, demonstrating that each component contributes beyond the other. Rules (deductive constraints) and examples (inductive patterns) thus address overlapping failure modes from different angles. Ordering matters: **GEAR**+2-shot outperforms 2-shot+**GEAR** by 3.3% on Qwen3-30B, supporting the "constrain-then-demonstrate" hypothesis where rules frame the action space first and examples demonstrate satisfying solutions. Notably, despite roughly 56% more input tokens, **GEAR**+2-shot generates fewer output tokens than **GEAR** alone (101 vs 114 on Qwen3-30B), yielding lower latency

[2]We use *Step Accuracy* (or *Step Pass@1*) to avoid confusion with end-to-end trajectory success in the agent literature; it measures per-step joint correctness of action name and parameters (Section 3).

since output dominates inference cost. We hypothesize that combined rule-and-example guidance increases generation confidence and prunes the decoding search space, yielding shorter traces; confirming this mechanism would require mechanistic interpretability and is left to future work (Appendix K). Component ablations on Qwen3-30B confirm that neither DirectiveOnly-**GEAR** (40.97%) nor PedagogyOnly-**GEAR** (41.61%) alone matches full **GEAR** (47.74%).

Comparison with Adaptation Methods. We compare **GEAR** against three stronger adaptation methods on Qwen3-4B (Tables 2 and 3). **GEAR**+2-shot outperforms LoRA SFT by 53.8%, RAG-3-shot by 71.4%, and ACE by 38.8% in Step Accuracy. Each baseline fails differently: RAG variants underperform even random 3-shot at both Qwen3-4B and Qwen3-30B, suggesting that within-domain query and trajectory embeddings carry limited discriminative signal. LoRA SFT peaks at epoch 2–4 before overfitting on the ∼18 examples per domain. ACE shows epoch-wise degradation (Table 3), which we attribute to context saturation as the playbook grows (generator input tokens scale 6.9M→10.6M across epochs). **GEAR**'s single-pass extraction sidesteps this failure mode while costing 4.1× fewer total tokens and 2.2× less wall-clock time than ACE (Appendix J.4). These results reinforce **GEAR**'s premise: with limited per-domain data, explicit rules abstract reasoning patterns that surface-level retrieval, gradient-based adaptation, and iterative prompt evolution all fail to capture.

Method	Action	Param.	Step
Zero-shot	57.53	37.57	26.45
3-shot (random)	68.09	57.41	45.15
RAG-3-shot ($\mathcal{R}_q$)	58.55	46.90	32.12
RAG-3-shot ($\mathcal{R}_{\text{full}}$)	59.74	46.30	32.84
LoRA SFT (best ckpt)	62.14	44.75	36.59
GEAR (ours)	72.51	53.55	45.48
GEAR+2-shot (ours)	**76.72**	**62.97**	**56.29**

Table 2: Training- and retrieval-based adaptation baselines on Qwen3-4B (Pass@1, %).

7 Analysis

7.1 Why Rules Outperform Alternatives?

The gaps to SFT, RAG, and ACE share a common cause: each operates on *surface features* (instance-level patterns for SFT, dense embeddings of q and $\mathcal{H}_t$ for RAG, accumulated playbook bullets for ACE) rather than underlying reasoning patterns. **GEAR**'s meta-reasoning extracts a different abstraction level: class-level failure patterns articulated as explicit constraints, decoupled from instance-specific surface form. The abstraction also shows in the gain distribution: **GEAR**'s improvements concentrate in Parameter Pass@1 over Action Pass@1 (+31.7% vs +26.6% over zero-shot on Qwen3-30B; +42.5% vs +26.0% on Qwen3-4B), with parameter formatting being a class-level regularity rules can encode explicitly. This suggests a practical boundary: failures with class-level structure (parameter formatting, retry and error handling) are rule-compressible, whereas cases that need concrete instance grounding are better served by exemplars—which is why combining rules with concise few-shot examples yields the strongest results rather than either alone.

Method	Action	Param.	Step
Zero-shot	57.10	37.10	26.80
ACE (epoch 1)	60.00	38.20	28.80
ACE (epoch 2)	55.60	39.90	29.30
ACE (epoch 3)	54.50	39.10	28.40
ACE (best playbook)	59.10	43.70	34.20
GEAR (ours)	66.06	41.66	32.85
GEAR+2-shot (ours)	**72.24**	**55.34**	**47.46**

Table 3: ACE vs **GEAR** on Qwen3-4B with matched Qwen3-4B extraction. Per-epoch rows show ACE's iteration trajectory; best playbook is per-API checkpoint selection. Cost breakdown in Appendix J.4.

7.2 Complexity Analysis

Across trajectory complexity (0–5 hops of prior API calls), all methods show degradation (step accuracy: 39–71% at hop 0 to 12–38% at hop 5), with **GEAR**+2-shot retaining the best tradeoff in performance, latency, and cost. Action space complexity (3–11 APIs per domain) shows no clear degradation pattern, suggesting API documentation provides sufficient guidance regardless of alternatives. Per-method curves for all three metrics in Appendix L.

	Qwen3-4B			Qwen3-30B		
Method	Action	Param.	Step	Action	Param.	Step
GEAR	72.51	53.55	45.48	72.81	56.25	47.74
GEAR+1-shot	69.50	53.94	44.59	70.74	55.39	46.44
GEAR+2-shot	**76.72**	**62.97**	**56.29**	**82.25**	**65.24**	**59.69**
GEAR+3-shot	71.96	59.16	48.68	71.99	57.49	48.48

Table 4: **GEAR**+k-shot scaling ablation (Pass@1, %).

8 Ablation Studies

Few-Shot Scaling with GEAR. We evaluate **GEAR**+k-shot for $k \in \{1, 2, 3\}$ on Qwen3-4B and Qwen3-30B (Table 4). Pure few-shot scaling plateaus by k=3 (47.78% on Qwen3-30B, only 4.7pp over 2-shot), but **GEAR**+k-shot is qualitatively different: non-monotonic rather than diminishing. **GEAR**+1-shot slightly underperforms **GEAR** alone (44.59% vs 45.48% on Qwen3-4B; 46.44% vs 47.74% on Qwen3-30B), **GEAR**+2-shot yields the largest jump (56.29% and 59.69%), and **GEAR**+3-shot regresses (48.68% and 48.48%). We do not claim that k=2 is optimal in general—only that, on this data, combining rules with examples has a sweet spot at two demonstrations. Our working hypothesis for the drop at k=3 is that additional examples dilute attention to the explicit rules; confirming this mechanism would require mechanistic interpretability analyses that we leave to future work.

Rule Components. Each rule comprises a directive and pedagogical component (Section 3.2). On Qwen3-30B, DirectiveOnly-**GEAR** achieves 40.97% step accuracy (+35.7% over zero-shot), capturing 61.4% of full **GEAR**'s improvement. PedagogyOnly-**GEAR** achieves 41.61% step accuracy (+37.8% over zero-shot), capturing 65.0% of the improvement (Table 1). Neither component alone matches full **GEAR** (47.74%), confirming their complementary roles: directives specify *what* to do and *when*, while the pedagogical component clarifies *how* through correct/incorrect behaviors.

Rule Format. Preliminary experiments showed that statement-style rules from $\mathcal{P}_{\text{meta-base}}$ yielded no improvement over zero-shot, while structured heuristics from $\mathcal{P}^*_{\text{meta}}$ (with explicit WHEN-THEN conditions, concrete examples, and verification criteria) produced consistent gains, motivating our iterative meta-prompt refinement (Table 5).

Contrastive Extraction. In preliminary experiments, we tested extracting rules from failed predictions only versus both successes and failures. Failure-only extraction caused 1–2% regression below zero-shot, as rules became overly restrictive and prohibited valid behaviors. Including successful predictions enables discriminative pattern identification, preventing such overfitting. This motivated including both outcomes in $\mathcal{D}_{\text{analysis}}$.

Statement-style ($\mathcal{P}_{\text{meta-base}}$) – No improvement
When an API call fails (timeout, 500 error, service unavailable), try alternative endpoints or different parameter values before giving up. If multiple functions fail consistently, proceed to Finish with an explanation.
Structured ($\mathcal{P}^*_{\text{meta}}$) – Consistent gains
WHEN API response contains 'status_code=503' or 'timeout'
THEN Try one alternative function if available; if alternative also fails, call Finish. Don't retry same function.
Wrong: Retried same function after 503.
Correct: Tried alternative once, then called Finish.

Table 5: Rule format comparison.

9 Conclusion

We introduced **GEAR**, a training-free framework that extracts compositional rules from tool-use failures via single-pass meta-reasoning and distills them into inference prompts. Across 78 API domains, **GEAR**+2-shot achieves the highest performance across all models and configurations tested: on Qwen3-30B it outperforms 3-shot by 24.9%, and on Qwen3-4B **GEAR** alone exceeds LoRA SFT by 24.3% and RAG-3-shot by 38.5%, while **GEAR**+2-shot beats the state-of-the-art context-engineering method ACE by 38.8% at 4.1× lower cost—validating our key hypothesis that deductive guidance (rules) and inductive learning (examples) provide complementary benefits. Remarkably, Qwen3-30B with **GEAR**+2-shot nearly doubles zero-shot accuracy while remaining under 1 s latency, demonstrating that explicit guidance enables rollout-free tool-use adaptation under resource-constrained deployment. **GEAR** offers a principled, rollout-free approach for real-time tool use when training, iterative rollouts, and extended test-time compute are infeasible.

10 Limitations

Static Rules in Dynamic Environments. **GEAR** assumes tool-use failures exhibit recurring patterns expressible as static natural-language rules, which holds well for structured API settings (parameter formatting, retry policies, predictable error states). The rule format in principle extends to any linguistically-describable failure (e.g., "WHEN a page element is not found after clicking, THEN retry with an alternative selector"), and skill libraries in open-ended environments like Minecraft (Wang et al., 2024) take a similar form. However, environments with non-stationary observations or high-variance interaction dynamics

(web UI drift, tool-schema changes, computer-use perception errors, long-horizon credit assignment) can stale previously extracted rules and shift the useful balance between rules and demonstrations: the k=2 sweet spot observed here need not transfer to a drifted regime. In such settings we recommend periodic re-extraction when failure modes change, and re-selecting k on a small held-out slice of the new regime rather than freezing the configuration found on static API data. Extending **GEAR** to open-ended, multimodal environments along these lines remains a promising direction for future work.

Cross-Domain Rule Transfer. **GEAR** can in principle extract cross-domain rules by jointly processing $\mathcal{D}_{\text{analysis}}$ from multiple domains, surfacing patterns shared across API categories. Naive transfer of domain-specific rules across domains with different API surfaces is not a fair test: WHEN/THEN conditions bind to API names, parameter schemas, and error states, so cross-domain application largely measures precondition mismatch rather than generalization of the underlying class-level pattern. In practice, even single-domain $\mathcal{D}_{\text{analysis}}$ saturates current context window limits given the length of multi-turn trajectories with comparative failure annotations, forcing per-domain extraction. Aggregating per-domain rules in a second-stage pass could surface transferable, domain-agnostic patterns without exceeding context windows, a promising direction for future work.

Per-Step Evaluation. Our evaluation uses per-step action prediction rather than full trajectory execution, forced by data availability: the APIs underlying ToolBench/ToolPreference are largely deprecated, preventing live trajectory execution, and no other public dataset provides both step-wise preference annotations and explicit failure cases (alternatives capture only clean optimal trajectories). LLM-as-judge filtering and trajectory history $\mathcal{H}_t$ across varying hop lengths mitigate single-path evaluation limitations, but per-step accuracy may not perfectly correlate with end-to-end success. Consistent improvements across 78 domains suggest robust findings; full-trajectory evaluation on live APIs remains future work.

11 AI Assistants in Research or Writing

We used Anthropic and OpenAI models throughout this work for paired programming, proofreading and refining manuscript text, brainstorming ideas for figures and tables, and generating visualization code. All AI-generated content was reviewed, verified, and edited by the authors, who retain full responsibility for the final manuscript.

References

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. 2025. Gepa: Reflective prompt evolution can outperform reinforcement learning. *Preprint*, arXiv:2507.19457.

Bhiman Kumar Baghel, Emma Jordan, Zheyuan Ryan Shi, and Xiang Lorraine Li. 2025. Resolving Under-Edit & OverEdit with iterative & neighbor-assisted model editing. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 14786–14808, Suzhou, China. Association for Computational Linguistics.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Carol Chen, Catherine Olsson, Christopher Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, Ethan Perez, and 32 others. 2022. Constitutional ai: Harmlessness from ai feedback. *Preprint*, arXiv:2212.08073.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michał Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. 2024. Graph of thoughts: solving elaborate problems with large language models. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'24/IAAI'24/EAAI'24. AAAI Press.

Zhenni Bi, Kai Han, Chuanjian Liu, Yehui Tang, and Yunhe Wang. 2025. Forest-of-thought: Scaling test-time compute for enhancing LLM reasoning. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 4253–4267. PMLR.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child,

Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, and 12 others. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Yuzheng Cai, Siqi Cai, Yuchen Shi, Zihan Xu, Lichao Chen, Yulei Qin, Xiaoyu Tan, Gang Li, Zongyi Li, Haojia Lin, Yong Mao, Ke Li, and Xing Sun. 2025. Training-free group relative policy optimization. *Preprint*, arXiv:2510.08191.

Sijia Chen, Yibo Wang, Yi-Feng Wu, Qing-Guo Chen, Zhao Xu, Weihua Luo, Kaifu Zhang, and Lijun Zhang. 2024. Advancing tool-augmented large language models: Integrating insights from errors in inference trees. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2024. Promptbreeder: self-referential self-improvement via prompt evolution. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, and 1 others. 2025. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA. Association for Computing Machinery.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. API-bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3102–3116, Singapore. Association for Computational Linguistics.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: iterative refinement with self-feedback. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Mistral. 2025. Introducing mistral 3. Published: 2025-12-2.

Shikhar Murty, Christopher D. Manning, Peter Shaw, Mandar Joshi, and Kenton Lee. 2024. Bagel: bootstrapping agents by guiding exploration with language. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Xuefei Ning, Zifu Wang, Shiyao Li, Zinan Lin, Peiran Yao, Tianyu Fu, Matthew B. Blaschko, Guohao Dai, Huazhong Yang, and Yu Wang. 2024. Can llms learn by teaching for better reasoning? a preliminary study. In *Advances in Neural Information Processing Systems*, volume 37, pages 71188–71239. Curran Associates, Inc.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. 2024. Optimizing instructions and demonstrations for multi-stage language model programs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 9340–9366, Miami, Florida, USA. Association for Computational Linguistics.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA. Curran Associates Inc.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2024. Gorilla: large language model connected with massive apis. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA. Curran Associates Inc.

Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023. CREATOR: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 6922–6939, Singapore. Association for Computational Linguistics.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, dahai li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *The Twelfth International Conference on Learning Representations*.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, Hong Kong, China. Association for Computational Linguistics.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessí, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: language models can teach themselves to use tools. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. 2025. Agentsquare: Automatic LLM agent search in modular design space. In *The Thirteenth International Conference on Learning Representations*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *Preprint*, arXiv:2402.03300.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Elias Stengel-Eskin, Archiki Prasad, and Mohit Bansal. 2024. Regal: refactoring programs to discover generalizable abstractions. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, and 150 others. 2025. Kimi k2: Open agentic intelligence. *Preprint*, arXiv:2507.20534.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2024. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*.

Zhangchen Xu, Adriana Meza Soria, Shawn Tan, Anurag Roy, Ashish Sunil Agrawal, Radha Poovendran, and Rameswar Panda. 2025. Toucan: Synthesizing 1.5m tool-agentic data from real-world mcp environments. *Preprint*, arXiv:2510.01179.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. 2024. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023a. Tree of thoughts: deliberate problem solving with large language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023b. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. 2026. Agentic context engineering: Evolving contexts for self-improving language models. In *The Fourteenth International Conference on Learning Representations*.

Qizheng Zhang, Michael Wornow, and Kunle Olukotun. 2025. Agentic plan caching: Test-time memory for fast and cost-efficient LLM agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and Yu Su. 2025. Skillweaver: Web agents can self-improve by discovering and honing skills. *Preprint*, arXiv:2504.07079.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2023. Large language models are human-level prompt engineers. In *The Eleventh International Conference on Learning Representations*.

A Extended Related Work

We provide an extended discussion of related work in this section, organizing the literature into training-based approaches, test-time reasoning methods, learning reusable components from experience, learning from failures and feedback, and prompt engineering techniques.

A.1 Tool-Augmented Language Models and Training Approaches

Large language models have been augmented with external tools through training-based approaches. Systems like Toolformer (Schick et al., 2023), ToolLLM (Qin et al., 2024), Gorilla (Patil et al., 2024), and API-Bank (Li et al., 2023) fine-tune models on tool-use data, enabling them to decide when and how to call APIs across diverse domains. Knowledge distillation approaches compress these capabilities from large models into smaller ones for efficient deployment. Recent work on the ToolPreference dataset (Chen et al., 2024) trains models on preference pairs using DPO (Rafailov et al., 2023), aligning them toward successful trajectories over failed ones. A more parameter-efficient alternative to full training is locate-and-edit model editing, which identifies knowledge-related parameters and updates only that subset (Baghel et al., 2025); however, these methods remain nuanced—they are mostly evaluated on factual knowledge benchmarks, and while the edit step itself can appear cheap, the causal-tracing prerequisite used to locate editable parameters is computationally expensive. While training-based and editing approaches achieve strong performance, they require access to model weights and substantial computational resources—unavailable to most practitioners under resource-constrained deployment. In contrast, **GEAR** operates entirely through prompting without any training, enabling broad applicability with open-weight models under limited compute.

A.2 Test-Time Reasoning and Search Methods

Test-time compute techniques enhance reasoning without training by allocating additional inference resources. ReAct (Yao et al., 2023b) improves performance through iterative self-reflection, where models reason about previous steps and adjust strategies. Tree-of-Thoughts (Yao et al., 2023a) and related multi-path exploration methods (Besta et al., 2024; Bi et al., 2025) generate multiple reasoning trajectories and select optimal paths through deliberate search. Chain-of-thought (CoT) prompting (Wei et al., 2022) and self-consistency (Wang et al., 2023) further enhance reasoning by eliciting intermediate steps and aggregating multiple reasoning paths. While effective, these methods operate on the current query in isolation, lacking access to cross-query failure patterns and recovery strategies from historical data. Consequently, they may repeatedly explore similar failure modes across different queries, increasing exploration overhead. **GEAR** complements these approaches by providing explicit guidance derived from failure distributions—reducing the search space by pruning known failure paths and prescribing effective recovery strategies. The combination of **GEAR**'s historical knowledge with test-time exploration offers a promising direction for efficient reasoning.

A.3 Learning Reusable Components from Experience

Recent work explores training-free extraction of reusable components from agent experience. Several approaches build libraries of executable code: SkillWeaver (Zheng et al., 2025) synthesizes skills as APIs for web agents, ReGAL (Stengel-Eskin et al., 2024) refactors programs into shared function libraries, and Voyager (Wang et al., 2024) constructs compositional skill libraries through embodied exploration in Minecraft. BAGEL (Murty et al., 2024) converts explored trajectories into demonstrations via round-trip refinement between instruction generation and execution. Agentic Plan Caching (Zhang et al., 2025) extracts and reuses plan templates from successful executions. Additional work includes CREATOR (Qian et al., 2023) for on-the-fly tool creation and AgentSquare (Shang et al., 2025) for modular agent architecture search. While these methods effectively extract reusable components through exploration and practice, they focus on capability acquisition, producing executable artifacts (code, APIs, plans, demonstrations). In contrast, **GEAR** addresses error prevention through systematic failure analysis, extracting behavioral constraint rules that prescribe what to avoid and how to recover from errors, complementing capability-focused approaches.

A.4 Learning from Failures and Feedback

Several approaches improve model performance by learning from errors and feedback. Reflexion (Shinn et al., 2023) generates verbal self-reflections after failed trajectories and replays tasks with these reflections in context. Self-Refine (Madaan et al., 2023) and Constitutional AI (Bai et al., 2022) similarly use self-critique for iterative improvement. RLHF (Ouyang et al., 2022) and preference optimization methods such as DPO (Rafailov et al., 2023) and GRPO (Shao et al., 2024) learn from comparative feedback to align model behavior. These methods differ fundamentally from **GEAR** in scope and reusability. Reflexion operates per-instance with trajectory-specific reflections requiring task replay, while RLHF and preference methods require model training. In contrast, **GEAR** performs cross-instance learning, extracting reusable compositional rules from multiple trajectories that transfer to new queries without replay or training, enabling amortized learning where extraction cost is incurred once but benefits all subsequent inferences.

A.5 Prompt Engineering and Optimization

Few-shot prompting guides model behavior by providing demonstration examples (Brown et al., 2020), enabling in-context learning without parameter updates. However, coverage is inherently sparse as k examples address k specific scenarios from a combinatorially large space, and models must inductively infer general patterns from specific instances. Retrieval-based methods (Lewis et al., 2020) improve demonstration relevance by dynamically selecting examples per query but add per-query latency while still relying on example-based learning. Automated prompt optimization methods such as APE (Zhou et al., 2023), OPRO (Yang et al., 2024), and evolutionary approaches (Fernando et al., 2024) refine prompts through meta-prompting or search algorithms. While effective at improving overall prompt quality, these methods treat prompts as monolithic black boxes, optimizing phrasing without extracting modular, interpretable components. In contrast, **GEAR** extracts explicit, compositional rules through systematic error analysis. Each rule addresses specific failure modes with quantified impact, enabling modular deployment and interpretability that black-box optimization cannot provide.

B Complete MDP Formalization

B.1 State Space

At each timestep t, the agent's state $s_t \in \mathcal{S}$ is defined as:

$$s_t = (\mathcal{I}, \mathcal{D}_{\text{API}}, q, \mathcal{H}_t) \tag{1}$$

System Instructions. The system instructions $\mathcal{I}$ specify the agent's role, output format requirements, and behavioral guidelines. This component is globally fixed across all queries and domains.

API Documentation. The API documentation $\mathcal{D}_{\text{API}} = \{(a_i, \text{doc}_i, \Theta_i)\}_{i=1}^{|\mathcal{A}|}$ provides structured information about available APIs, where a_i is the API identifier, doc_i is a natural language description, and Θ_i is the parameter space. This component is fixed per domain.

User Query. The user query $q \in \mathcal{Q}$ is a natural language input specifying the task to be accomplished. This component remains fixed throughout an episode.

Trajectory History. The trajectory history $\mathcal{H}_t = \langle (\alpha_0, o_0), \ldots, (\alpha_{t-1}, o_{t-1}) \rangle$ records all previous action-observation pairs, where α_k denotes the action at timestep k and o_k denotes the corresponding observation. At the initial timestep, the history is empty: $\mathcal{H}_0 = \langle\rangle$.

B.2 Action Space

At each timestep, the agent generates an action $\alpha_t \in \mathcal{A}$, where $\alpha_t = (a_t, \theta_t)$ consists of an API identifier $a_t \in \{a_1, \ldots, a_{|\mathcal{A}|}\}$ and parameters $\theta_t \in \Theta_{a_t}$. The parameter space Θ_i for each API a_i may contain discrete, continuous, or structured parameters.

B.3 Objective Functions

Deployment Objective. The agent follows a policy $\pi : \mathcal{S} \to \mathcal{A}$ that maps states to actions. The deployment objective is to maximize expected task completion over the query distribution $\mathcal{Q}$:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{q \sim \mathcal{Q}} \mathbb{E}_{\tau \sim p(\cdot|q,\pi)}[r(\tau)] \tag{2}$$

where τ denotes a complete trajectory and $r(\tau)$ is a binary reward based on task completion.

Evaluation Objective. We evaluate policies using per-step action accuracy on a dataset of optimal trajectories $\mathcal{D}$:

$$\text{Accuracy}(\pi) = \mathbb{E}_{(s,\alpha^*) \sim \mathcal{D}}[\mathbb{1}[\pi(s) = \alpha^*]] \tag{3}$$

Two actions are considered equal if both components match: $(a, \theta) = (a', \theta') \iff (a = a') \wedge (\theta = \theta')$.

C Baseline Policy Formulations

Contemporary Large Reasoning Model (LRM) policies implement π through in-context learning, varying in how they augment the agent's state with guidance information.

C.1 Policy Formulations

Zero-Shot Policy. The zero-shot policy operates on the standard state representation:

$$s_t = (\mathcal{I}, \mathcal{D}_{\text{API}}, q, \mathcal{H}_t) \tag{4}$$

relying entirely on the model's pre-trained knowledge and reasoning capabilities.

Few-Shot Policy. The few-shot policy augments the state with demonstration examples:

$$s_t = (\mathcal{I}, \mathcal{D}_{\text{API}}, \mathcal{E}, q, \mathcal{H}_t) \tag{5}$$

where the example set $\mathcal{E} = \{(q^{(i)}, \mathcal{H}^{(i)})\}_{i=1}^{k}$ contains k query-trajectory pairs demonstrating successful interactions. This approach requires inductive reasoning to generalize patterns from specific examples.

Chain-of-Thought Policy. Chain-of-Thought (CoT) augments the system instructions $\mathcal{I}$ with explicit reasoning directives, encouraging step-by-step problem decomposition at the cost of increased output tokens.

Rule-Augmented Policy (GEAR). **GEAR** augments the state with explicit behavioral rules:

$$s_t = (\mathcal{I}, \mathcal{D}_{\text{API}}, \mathcal{R}, q, \mathcal{H}_t) \tag{6}$$

where the rule set $\mathcal{R} = \{r_1, \ldots, r_{|\mathcal{R}|}\}$ contains domain-specific constraints extracted through meta-reasoning. Each rule $r_i = (\text{Directive}_i, \text{Pedagogical}_i)$ comprises: (1) a directive specifying the condition and required behavior (WHEN-THEN structure), and (2) a pedagogical component illustrating correct behavior, incorrect behavior, and test criteria for verification. Unlike few-shot learning which requires inductive pattern inference, both rule components enable deductive constraint application.

Combined Policy (GEAR+Few-Shot). **GEAR** can be combined with few-shot examples:

$$s_t = (\mathcal{I}, \mathcal{D}_{\text{API}}, \mathcal{R}, \mathcal{E}, q, \mathcal{H}_t) \tag{7}$$

This formulation leverages both deductive reasoning (via rules $\mathcal{R}$) and inductive reasoning (via examples $\mathcal{E}$), enabling complementary guidance mechanisms.

D Meta-Prompt Materials

This appendix provides the prompts used in **GEAR**'s meta-prompt creation stage (Section 4.1).

Algorithm 1 formalizes the two-stage process. Stage 1 generates the rubric $\mathcal{C}$ that anchors all subsequent evaluation, ensuring critique criteria remain fixed across iterations. Stage 2 alternates between refinement (the LRM rewrites the prompt based on prior critique) and self-critique (the LRM evaluates the rewrite against $\mathcal{C}$), terminating when the maximum rubric grade is achieved or the iteration budget K is exhausted. The optional human-validation step at the end performs a structural sanity check (input placeholders, output schema, parser compatibility), which we found necessary because LRMs occasionally produce malformed JSON or mismatched delimiters despite high self-assessment grades.

D.1 Base Meta-Prompt

The base meta-prompt $\mathcal{P}_{\text{meta-base}}$ (Figure 3) serves as the starting point for iterative refinement. This initial version produces statement-style rules that lack structured WHEN-THEN format.

D.2 Stage 1: Rubric Generation and Critique Prompt

In Stage 1, an LRM generates an evaluation rubric $\mathcal{C}$ and critiques $\mathcal{P}_{\text{meta-base}}$ against this rubric. The critique prompt (Figures 4, 5) instructs the LRM to create comprehensive evaluation criteria and perform detailed analysis.

D.3 Stage 2: Self-Critique Prompt

In Stage 2, the same LRM iteratively refines the prompt using self-critique. At each iteration k, the LRM generates an improved prompt $\mathcal{P}_k$ based on previous critique c_{k-1}, then evaluates $\mathcal{P}_k$ against the fixed rubric $\mathcal{C}$.

Refinement Prompt. The refinement prompt (Figures 6, 7) takes the previous prompt $\mathcal{P}_{k-1}$ and critique c_{k-1} as input:

Self-Critique Prompt. After generating $\mathcal{P}_k$, the LRM evaluates it against rubric $\mathcal{C}$. The self-critique prompt (Figures 8, 9) explicitly instructs the LRM to be "harsh" and avoid "grade inflation," as we observed LRMs tend toward overconfidence in self-assessment. The prompt requires specific evidence from the generated prompt for each criterion and a detailed explanation of any grade below A+.

D.4 Optimization Trajectory and Rule Format Validation

Table 6 traces the structural evolution of the meta-prompt across the five optimization iterations. The base $\mathcal{P}_{\text{meta-base}}$ received a D- rubric grade, with optimization converging to an A+ grade at iteration 4. Two qualitative shifts drive the improvement: (i) the analysis stage evolves from unstructured rule extraction to an 8-step structured pipeline, and (ii) rule outputs transition from free-form statements to a structured WHEN-THEN format with directive and pedagogical components. Light formatting corrections during human validation (mismatched braces, Python-specific JSON parsing issues) produced the final $\mathcal{P}^*_{\text{meta}}$. Intermediate prompts along the optimization trajectory ($\mathcal{P}_1$–$\mathcal{P}_4$) are omitted here for space; readers who need them may contact the corresponding author.

To empirically validate that the structured rule format produced by $\mathcal{P}^*_{\text{meta}}$ outperforms the statement-style format from $\mathcal{P}_{\text{meta-base}}$, we ran both rule sets through the same evaluation pipeline (Table 5). Statement-style rules yielded no improvement over zero-shot, while structured rules produced consistent gains, confirming that the structural transformations in Table 6 are not cosmetic—they translate directly into evaluation gains.

D.5 Optimized Meta-Prompt

The optimized meta-prompt $\mathcal{P}^*_{\text{meta}}$ (Figure 10) is the final output after self-critique refinement and human validation. This prompt produces structured rules with explicit WHEN-THEN directives and pedagogical components.

The meta-prompt comprises eight structured sections that guide the LRM through: (1) role and task definition, (2) task context and baseline metrics, (3) data structure explanation, (4) systematic analysis steps including failure categorization, pattern identification, and root cause analysis, (5) rule extraction with structured output format, (6) validation criteria, (7) meta-analysis of coverage and expected improvement, and (8) final output format specification.

D.5.1 Key Design Principles

The meta-prompt incorporates several design principles that emerged from the self-critique process described in Section 4.1. The complete prompt is shown in Figure 10; we summarize its key structural components and design rationale here.

Structured Multi-Step Analysis Framework. The prompt enforces a systematic eight-step process that prevents premature rule extraction. The LRM must first complete comprehensive failure analysis (Steps 1-4) before synthesizing rules (Steps 5-6), followed by validation and meta-analysis (Steps 7-8). Each step produces structured output that serves as input to subsequent steps, ensuring thorough coverage. For instance, Step 1 requires frequency counts for each failure type (e.g., "Wrong function called: X examples"), which Step 4 then analyzes for root causes.

Explicit Categorization Schema. The prompt defines three primary failure categories: action selection errors, parameter population errors, and API error handling failures, with multiple subcategories under each. This taxonomy guides the LRM to systematically organize failures rather than analyzing them in an ad-hoc manner. For example, parameter errors are subdivided into format mismatches, missing values, wrong data types, and documentation misinterpretation, each requiring different rule types.

Frequency-Based Filtering. The prompt explicitly instructs the LRM to extract rules only for patterns occurring in $\geq \tau$ examples, preventing overfitting to rare edge cases. The LRM must track occurrence counts during analysis and justify each rule with its frequency. This threshold-based approach balances coverage (addressing common issues) with generalization (avoiding instance-specific rules).

Contrastive Learning Guidance. Step 5 requires analyzing successful predictions alongside failures to identify what distinguishes correct from incorrect behaviors. The prompt provides a structured template for this comparison answering what failed, what successful examples did right and Actionable mitigation strategy. This contrastive analysis prevents rules that would regress correct predictions, addressing the issue identified in Section 8.

Generalization Through Examples. The prompt includes just a couple of explicit good vs. bad rule examples to guide abstraction. These examples demonstrate the desired level of abstraction—rules should capture structural patterns rather than surface features.

Structured Output Schema. The prompt fully specifies the JSON schema (detailed in Table 7) for extracted rules, including all required fields and their formats. This structured output ensures rules are parseable, categorizable, and contain sufficient

Meta-Prompt	Structured Analysis	Rule Format	Key Changes
$\mathcal{P}_{\text{meta-base}}$	No	Statement	Starting point (D- grade); direct rule extraction with no structured analysis pipeline; rule format unspecified.
$\mathcal{P}_1$	No	Statement	Introduced categorization of rules into action-specific and parameter-specific types; other features unchanged.
$\mathcal{P}_2$	Yes	Structured	Proposed a 5-step analysis pipeline; rules reformatted into a structured WHEN-THEN representation.
$\mathcal{P}_3$	Yes	Structured	Extended to a 6-step analysis pipeline with refined structured rule formatting.
$\mathcal{P}_4$	Yes	Structured	Further extended to an 8-step analysis pipeline; converged to A+ rubric grade.
$\mathcal{P}^*_{\text{meta}}$	Yes	Structured	Final version after light formatting corrections during human validation (resolving mismatched braces and JSON parsing issues).

Table 6: Structural evolution of the meta-prompt across optimization iterations. The rubric grade improved from D- ($\mathcal{P}_{\text{meta-base}}$) to A+ by iteration four ($\mathcal{P}_4$), after which the maximum grade was achieved. $\mathcal{P}^*_{\text{meta}}$ denotes the final version used in all reported experiments.

information for validation and deployment decisions.

Multi-Level Validation Framework. Step 7 provides explicit validation criteria organized as yes/no checks:

- *Coverage*: Does this rule prevent failures in $\geq \tau$ examples?
- *Regression*: Could this rule break currently correct predictions?
- *Atomicity*: Does this rule address exactly one failure pattern?
- *Implementability*: Can this be added to a prompt as a clear instruction?
- *Testability*: Is the test criterion measurable and unambiguous?
- *Generalizability*: Does this apply across different APIs, not just one?
- *Consistency*: Does this conflict with other rules?

The prompt requires the LRM to evaluate each extracted rule against all criteria and revise or remove rules that fail any check.

Meta-Analysis and Coverage Estimation. Step 8 requires the LRM to estimate the expected improvement from extracted rules by computing: (1) total failures in dataset, (2) failures addressed by rules (sum of estimated impacts), (3) coverage percentage, and (4) predicted performance gains. This meta-analysis serves as a self-consistency check—if coverage is low (<60%) or predicted improvement is implausibly high (>50%), the LRM must identify gaps or reconsider rule quality.

Domain-Agnostic Parameterization. The prompt uses placeholders for domain-specific information: `{api_documentation}`, `{system_instruction}`, `{baseline_metrics}`, and `{examples}`. All instructions and analysis frameworks are written generically to work across diverse API domains without modification. This design enables the single meta-prompt to be reused across all 78 evaluation domains.

E Extract Phase Details

This section provides details on the Extract phase (Section 4.3), including the complete JSON schema Table 7 for extracted rules, an extracted rule in JSON (Figure 19) and then serialized (Figure 20) format for usage in next phase, and representative examples from BMR/TMR (Table 8) and finance (Table 9) domains.

F Data Filtering

The original ToolPreference dataset contains approximately 33,000 single-turn examples constructed by tracing successful paths in depth-first search decision trees. Specifically, we use `dpo_preferencepairs_train.json`, which contains user queries, action histories, and preferred/dispreferred next-action tuples. This construction process resulted in preference pairs where some "preferred" actions were suboptimal recovery steps rather than truly optimal next actions.

To address this noise, we employed LLM-as-judge filtering using Qwen3-30B to verify whether either action in each preference tuple represents the optimal next action given the current state. Specifically, the judge evaluates each tuple and classifies it as: (1) preferred action is optimal, (2) dispreferred action is optimal, or (3) neither action is

Field	Type	Required	Description
`rule_id`	integer	Yes	Unique identifier for the rule (1, 2, 3, ...)
`name`	string	Yes	Short descriptive name (e.g., "HTML Error Detection")
`priority`	string	Yes	Priority level: "High", "Medium", or "Low". Determines deployment order.
`category`	string	Yes	Failure category: "Action Selection", "Parameter Population", or "Error Handling"
`addresses`	string	Yes	Specific failure type and frequency (e.g., "Repeated failed API calls (12 examples)")
`condition`	string	Yes	WHEN clause: boolean predicate specifying when rule applies (e.g., "When API response starts with '<!DOCTYPE'")
`action`	string	Yes	THEN clause: prescribed behavior (e.g., "Recognize service unavailability, try alternative once, then call Finish")
`example_wrong`	string	Yes	Concrete example of incorrect behavior from data (e.g., "Retried bmr_index 3 times after HTML error")
`example_correct`	string	Yes	Concrete example of correct behavior (e.g., "After HTML error, tried tmr_index once, then called Finish")
`test_criterion`	string	Yes	Measurable verification method (e.g., "Check if model stops after HTML error and tries alternative")
`estimated_impact`	string	Yes	Expected improvement as percentage (e.g., "12% of action failures")

Table 7: Complete JSON schema for extracted rules. Each rule object must contain all required fields.

Rule ID	Category	Summary
1	Error Handling	**HTML Error Detection**: When response contains HTML tags, recognize as service unavailability. Do not retry same function. Try alternative once or call Finish.
2	Parameter Population	**Boolean String Format**: When parameter name suggests boolean (e.g., "inimperial"), check API docs for exact format. Use string "true"/"false", not synonyms like "metric"/"imperial".
3	Action Selection	**Direct Calculation**: When user provides all required parameters (age, weight, height, sex), call calculation API directly. Do not call helper APIs to retrieve values.

Table 8: Example rules extracted from BMR/TMR domain.

optimal. Tuples classified as (3) were discarded, while tuples classified as (2) had their preference labels swapped. This filtering was critical for extracting meaningful behavioral rules rather than surface-level patterns from noisy supervision.

After filtering, the dataset was reduced from approximately 33,000 to 1,949 high-quality examples suitable for rule extraction. The complete filtering prompt is shown in Figure 21.

G Dataset Statistics

From the filtered dataset, we selected domains containing $\geq$20 valid examples to ensure sufficient data for both rule extraction and evaluation. This threshold yielded 78 domains with example counts ranging from 20–45 per domain. Domains span diverse categories including health (BMR/TMR calculation), finance (stock data), search, and utilities. We split the 78 domains 70:30 into $\mathcal{D}_{\text{extract}}$ (n=1,397) for rule extraction and $\mathcal{D}_{\text{test}}$ (n=552) for evaluation. The per-domain distribution of examples across both subsets is visualized in Figure 23. Additionally, Table 10 presents the dataset distribution grouped by action space size $|\mathcal{A}|$, where each row aggregates domains sharing the same number of available APIs in their documentation $\mathcal{D}_{\text{API}}$.

H Model Specifications

Table 11 provides specifications for all models used in our experiments. We evaluate three open-source instruct-tuned models: Qwen3-30B (Yang et al., 2025), Qwen3-4B (Yang et al., 2025), and Ministral3-14B (Mistral, 2025). All models

Rule ID	Category	Summary
1	Error Handling	**Empty Response Handling**: When API returns empty list/dict, check if this is valid result (no data found) vs error. If unclear, call alternative data source or Finish with explanation.
2	Parameter Population	**Stock Symbol Format**: Extract exact ticker symbols from user query (e.g., "AAPL", "GOOGL"). Do not convert to full company names. Preserve uppercase format.
3	Action Selection	**Time Range Selection**: When query mentions time period (e.g., "last 2 weeks"), use time-range API accepting start/end dates rather than calling daily APIs repeatedly.

Table 9: Example rules extracted from Finance domain.

	$\mathcal{D}_{\text{extract}}$		$\mathcal{D}_{\text{test}}$	
$\lvert\mathcal{A}\rvert$	**#Domains**	**#Examples**	**#Domains**	**#Examples**
3	17	303	17	120
4	14	248	14	100
5	13	244	13	97
6	6	106	6	42
7	10	181	10	70
8	6	98	6	37
9	2	46	2	18
10	1	18	1	7
11	9	153	9	61
Total	**78**	**1,397**	**78**	**552**

Table 10: Dataset composition by action space size. $\lvert\mathcal{A}\rvert$ denotes the number of available APIs in $\mathcal{D}_{\text{API}}$ for each domain.

are evaluated with greedy decoding (temperature T=0).

Local Deployment. Open-source Qwen3 models and Ministral3-14B were served locally using vLLM (Kwon et al., 2023) with the following commands:

- **Qwen3-30B:**[3] `vllm serve Qwen3-30B-A3B-Instruct-2507 -dtype auto -tensor-parallel-size 4`
- **Qwen3-4B:**[4] `vllm serve Qwen3-4B-Instruct-2507 -dtype auto -max_model_len 10000 -tensor-parallel-size 1`
- **Ministral3-14B:**[5] `vllm serve Ministral-3-14B-Instruct-2512 -dtype auto -max_model_len 10000 -tensor-parallel-size 2`

[3] https://huggingface.co/Qwen/Qwen3-30B-A3B-Instruct-2507

[4] https://huggingface.co/Qwen/Qwen3-4B-Instruct-2507

[5] https://huggingface.co/mistralai/Ministral-3-14B-Instruct-2512

Hardware. Local inference was conducted on a server with 8× NVIDIA A100-SXM4-40GB GPUs (320GB total VRAM), CUDA 12.8, and driver version 570.124.06. Qwen3-30B utilized 4 GPUs via tensor parallelism, while Qwen3-4B ran on a single GPU.

I Baseline Configurations

This section details the prompt configurations for all baselines and **GEAR** variants evaluated in our experiments.

Zero-shot. Uses the standard ReAct prompt containing system instructions $\mathcal{I}$, API documentation $\mathcal{D}_{\text{API}}$, user query q, and trajectory history $\mathcal{H}_t$. No additional instructions or examples are provided. See Figure 24.

Zero-shot-CoT. Extends the zero-shot prompt by adding an explicit reasoning instruction to the task section: *"Think step by step, examine previous steps and thoughts, then decide."* See Figure 25.

Few-shot (1/2/3-shot). Includes randomly sampled examples from $\mathcal{D}_{\text{extract}}$, where each example contains the full user query, trajectory history, and action specification. The same prompt template is used for all few-shot variants; only the number of examples varies. See Figure 26.

GEAR. Incorporates extracted rules as guidelines positioned before the user query and history, as described in Section 3.2 in zero-shot prompt. See Figure 27.

GEAR + 2-shot. Combines rules with two in-context examples. Examples are placed *between* the rules and the user query. See Figure 28.

2-shot + GEAR. An alternative ordering where two in-context examples are placed *before* the guidelines/rules section.

Model	Version / Endpoint	Access	Temp	Max Tokens
Qwen3-30B	`Qwen3-30B-A3B-Instruct-2507`	Local (vLLM)	0	500
Qwen3-4B	`Qwen3-4B-Instruct-2507`	Local (vLLM)	0	500
Ministral3-14B	`Ministral-3-14B-Instruct-2512`	Local (vLLM)	0	500

Table 11: Model specifications and configurations. All models use greedy decoding (T=0) and were hosted locally using vLLM (Kwon et al., 2023).

J Adaptation Baseline Details

J.1 LoRA SFT Configuration

We fine-tune Qwen3-4B on $\mathcal{D}_{\text{extract}}$ using LoRA adapters with rank r=16, α=32, dropout 0.05, applied to all linear layers in the attention and MLP blocks. Training uses AdamW with learning rate 5×10^{-5}, batch size 16, and a cosine schedule with 5% warmup. Each training example formats the trajectory state s_t as input and the gold next action α_t^* as output, following the same ReAct format used by all other baselines. We train for 10 epochs and select the best checkpoint based on step accuracy on $\mathcal{D}_{\text{extract}}$. With approximately 18 examples per domain across 78 domains, models typically peak at epoch 2–4 before exhibiting overfitting: parameter accuracy degrades faster than action accuracy, consistent with fine-tuning latching onto instance-specific parameter patterns rather than transferable reasoning.

J.2 RAG-3-shot Configuration

We use `all-MiniLM-L6-v2` (Reimers and Gurevych, 2019) to encode candidate demonstrations from $\mathcal{D}_{\text{extract}}$. Retrieval is performed within each API domain to ensure demonstrations come from the same toolset as the query. For the $\mathcal{R}_q$ variant, only the user query q is encoded; for $\mathcal{R}_{\text{full}}$, we concatenate q with the realized trajectory history $\mathcal{H}_t$ (truncated to 512 tokens for the encoder). Cosine similarity ranks candidates, and the top-3 are formatted as standard few-shot demonstrations identical in format to the random 3-shot baseline. Both variants underperform random selection at both Qwen3-4B and Qwen3-30B (Section 6), suggesting that within a single API domain, query and trajectory embeddings provide limited discriminative signal.

J.3 ACE Implementation Details

We adapted the official ACE codebase to the Tool-Preference dataset following the published extension guidelines. The default configuration produced near-zero-shot results; we identified and corrected two issues. First, the default setting of multiple reflector rollouts (`--max_num_rounds`) retained only the last reflection while discarding earlier reflections that often correctly identified the problem; we set rollouts to 1. Second, the curator (with `--curator_freq 3`) considered only the last data point's reflection due to a bug; we patched the curator to retain all accumulated reflections. Setting `--curator_freq 1` subsequently inflated the playbook with redundant bullets and degraded performance, so we retained the default frequency. The results in Section 6 reflect ACE after these corrections, with hyperparameters tuned for best end-to-end performance. We ran Qwen3-4B for all three roles (Generator, Reflector, Curator) over 3 epochs on all 78 APIs and report the best playbook (selected per-API on $\mathcal{D}_{\text{extract}}$).

J.4 ACE Cost Breakdown

This appendix provides detailed cost accounting for the **GEAR** vs. ACE comparison in Section 6. All numbers are aggregated across the 78 API domains in $\mathcal{D}_{\text{extract}}$. For fair comparison, both methods use Qwen3-4B for all components (ACE's Generator/Reflector/Curator roles; **GEAR**'s rule extraction).

ACE Training Cost. ACE training runs for 3 epochs over all 78 APIs, with each epoch comprising Generator (executes the task), Reflector (analyzes errors), and Curator (updates the playbook) phases. Generator input tokens grow across epochs as the playbook accumulates (6.86M $\rightarrow$ 8.82M $\rightarrow$ 10.62M), reflecting the context saturation we identify as the source of ACE's epoch-wise performance degradation (Section 6).

GEAR Training Cost. **GEAR**'s training pipeline is a single-pass three-step process: (1) generate zero-shot baseline predictions on $\mathcal{D}_{\text{extract}}$ to populate $\mathcal{D}_{\text{analysis}}$, (2) extract rules via $\mathcal{P}^*_{\text{meta}}$, and (3) select the best rule set per domain based on $\mathcal{D}_{\text{extract}}$ performance. Steps 2 and 3 are run three

	Epoch 1		Epoch 2		Epoch 3		
Phase	In	Out	In	Out	In	Out	**Total**
Generator	6,860	375	8,820	410	10,617	424	27,506
Reflector	2,572	455	2,594	458	2,607	462	9,148
Curator	1,787	117	2,039	109	2,256	112	6,420
Total	**11,220**	**947**	**13,453**	**976**	**15,481**	**998**	**43,074**
Time (h)	3.47		3.59		3.69		**10.75**

Table 12: ACE training cost across 3 epochs (Qwen3-4B). Token counts in thousands (K). Generator input grows across epochs as the playbook accumulates.

times to mitigate non-determinism under greedy decoding. Total training cost is 10.2M tokens and 5.27 hours.

Step	**Input** (K tok)	**Output** (K tok)	**Total** (K tok)	**Time** (h)
1: Zero-shot baseline	1,799	116	1,915	0.30
2: Rule extraction ($\times$3)	3,633	967	4,600	4.42
3: Run selection ($\times$3)	3,521	149	3,670	0.55
Total	**8,954**	**1,232**	**10,185**	**5.27**

Table 13: **GEAR** training cost (single-pass, Qwen3-4B extractor). Steps 2 and 3 are repeated three times to mitigate non-determinism in greedy decoding.

Test Cost. At inference, **GEAR**'s lower input token count reflects its compact rule set ($\sim$5–15 rules per domain) compared to ACE's accumulated playbook ($\sim$6M generator input tokens per pass over $\mathcal{D}_{\text{test}}$). **GEAR**+2-shot adds only $\sim$600K input tokens over **GEAR** alone, while exceeding ACE's best playbook by 38.8% step accuracy.

Method	**Input** (K tok)	**Output** (K tok)	**Total** (K tok)	**Time** (h)
ACE (best playbook)	6,133	292	6,424	1.20
GEAR	1,160	49	1,209	0.18
GEAR+ 2-shot	1,756	58	1,814	0.22

Table 14: Test-time inference cost on $\mathcal{D}_{\text{test}}$ (Qwen3-4B evaluator).

Summary. Across the full pipeline (training + test), **GEAR**+2-shot uses 12.0M tokens and 5.49 hours, versus ACE's 49.5M tokens and 11.95 hours—a 4.1$\times$ token and 2.2$\times$ time efficiency advantage at substantially higher step accuracy.

K Extended Results

Table 15 presents complete results across open-model configurations. We highlight key findings below. All results are discussed in terms of Step Pass@1.

Zero-shot-CoT Ineffectiveness. Across open models, Zero-shot-CoT provides only slight gains: $+1.6\%$ on Qwen3-30B, $+6.1\%$ on Qwen3-4B, and $+5.2\%$ on Ministral3-14B. This confirms that generic reasoning prompts offer limited value for structured tool-calling compared to explicit rule guidance, even for instruction-tuned open-source models.

Consistent GEAR Effectiveness Across Models. **GEAR** or its variants match or exceed the strongest few-shot baselines across nearly all configurations tested. On Qwen3-30B and Qwen3-4B, **GEAR**+2-shot substantially outperforms 3-shot (by 24.9% and 24.7% respectively). Ministral3-14B achieves best results with **GEAR** alone, exceeding 3-shot by 10.7%.

Ministral3-14B: Reversed Complementarity. Ministral uniquely achieves best Step with **GEAR** alone (42.53%), outperforming both **GEAR**+2-shot (38.49%) and 3-shot (38.42%). The ordering effect is also reversed: 2-shot+**GEAR** outperforms **GEAR**+2-shot by 7.4%. We hypothesize that this smaller model experiences interference when processing both rules and examples simultaneously, with rules alone providing cleaner guidance.

Ordering Effect. The "constrain-then-demonstrate" ordering (**GEAR**+2-shot) outperforms "demonstrate-then-constrain" (2-shot+**GEAR**) on Qwen3-30B (3.3%) and Qwen3-4B (18.6%), with amplified effects on the smaller model. The exception is Ministral (reversed by 7.4%), indicating model-specific sensitivity to prompt structure.

Efficiency and Generation Confidence. **GEAR**+2-shot achieves lower latency than **GEAR** alone despite higher input token counts on Qwen3-30B (0.78s vs 0.85s) and Qwen3-4B (1.13s vs 1.17s). This supports the hypothesis that combined rule-example guidance increases generation confidence, reducing the model's search space during decoding—evidenced by fewer output tokens for **GEAR**+2-shot than **GEAR** alone (101 vs 114 on Qwen3-30B; 108 vs 121 on Qwen3-4B).

Future Directions. The model-specific variations observed in **GEAR**'s effectiveness, particularly the reversed complementarity in Ministral, raise questions about the internal reasoning mecha-

nisms underlying tool-calling. Understanding why certain models benefit more from deductive rules versus inductive examples, or why prompt ordering effects vary across architectures, could yield deeper insights into how these models process structured generation tasks. We leave such interpretability investigations, including attention pattern analysis and probing studies on open-weight models, for future research.

L Complexity Analysis

We analyze how performance varies with task complexity along two dimensions: trajectory complexity (number of hops) and action space complexity (number of available APIs).

L.1 Trajectory Complexity (Hops)

We operationalize trajectory complexity as the number of previous API calls (hops) in the trajectory history $\mathcal{H}_t$. Higher hop counts indicate more complex multi-step reasoning tasks. Figure 29 presents results on Qwen3-30B.

Key Findings.

- All methods exhibit performance degradation as complexity increases, with Step Pass@1 declining from 39-71% at hop 0 to 12-38% at hop 5
- **GEAR**+2-shot maintains highest performance across all complexity levels (70.71% at hop 0, 37.5% at hop 5)
- Relative ordering of methods remains consistent regardless of complexity, suggesting complementarity between rules and examples is robust to task difficulty
- Parameter prediction degrades more severely than action selection: zero-shot Parameter Pass@1 drops from 45.45% to 12.5%, while Action Pass@1 remains stable (74.75% to 75.0%)
- Latency and cost increase approximately linearly with hop count due to longer context windows
- Critical-**GEAR**offers favorable efficiency, maintaining competitive accuracy with lower cost than **GEAR** and **GEAR**+2-shot across all complexity levels

L.2 Action Space Complexity (Number of APIs)

We operationalize action space complexity as the number of available APIs per domain, ranging from 3 to 11 in our dataset. Larger action spaces present more alternatives for the model to choose from, potentially increasing decision difficulty. Figure 30 presents results on Qwen3-30B.

Key Findings.

- Unlike trajectory complexity, performance does not exhibit a clear monotonic degradation pattern with increasing action space size
- Performance varies considerably across API counts but appears more influenced by domain-specific characteristics than action space size alone
- **GEAR**+2-shot maintains consistently strong performance across action space sizes, achieving highest Step Pass@1 at most configurations (e.g., 61.86% at 5 APIs, 49.18% at 11 APIs)
- Latency remains relatively stable across action space sizes, with occasional outliers due to domain-specific factors
- Cost increases moderately with API count due to longer API documentation in the prompt
- Critical-**GEAR**achieves competitive performance with substantially lower cost across all action space sizes

These results suggest that action selection difficulty is less sensitive to the number of available alternatives than to trajectory length, likely because API documentation provides sufficient discriminative information regardless of action space size.

Base Meta-Prompt $\mathcal{P}_{\text{meta-base}}$

```
You are an expert prompt engineer and a brilliant systems analyst. I have run a baseline
experiment on tool-use tasks using a powerful LLM.

The results show a clear reasoning gap:
- **Action Pass@1 (Choosing the right tool)**: {action_pass_rate}%
  ({action_pass_count}/{total})
- **Parameter Pass@1 (Populating the tool's input)**: {param_pass_rate}%
  ({param_pass_count}/{total})
- **E2E Pass@1 (Both correct)**: {e2e_pass_rate}% ({e2e_pass_count}/{total})

The "Parameter" failures are *semantic*—the model picks the wrong *values* for the parameters,
even when the syntax is correct.

The model is failing at two levels:
1. **Action Reasoning:** It struggles to select the correct tool from the API list.
2. **Parameter Reasoning:** It struggles to populate parameters with correct values based on
  the user's query and conversation history.

Your task is to analyze {num_samples} examples and generate 5-7 general-purpose, high-impact
"Heuristic Reasoning Rules" that will help the model improve *both* action selection and
parameter population.

---
**GUIDELINES FOR RULES:**

* **Target Both Reasoning Failures:** Generate rules that guide *Action Selection* and rules
  that guide *Parameter Population*.
* **Go Deep, Not Surface-Level:** The rules must be generalizable. Do not create rules that
  only apply to a single example or task.
    * **BAD RULE (Surface-Level):** "If the user asks about 'NFLX' and 'AMZN', use the
      `ranges_2w_for_finance...` tool."
    * **GOOD RULE (Structural Action Rule):** "Before acting, check the conversation
      history. If a previous action failed, do not repeat the exact same action. Instead,
      select a different tool or modify the parameters."
    * **GOOD RULE (Structural Parameter Rule):** "When an API has multiple parameters, scan
      the user's query and history for specific values for each one. Do not use example
      values from the documentation."
---

Here are the {num_samples} examples:

{examples}

---

Based on your analysis of all cases, what are the 5-7 general-purpose reasoning rules that will
improve both the `Action Pass@1` and `Parameter Pass@1` scores?

**Distilled Heuristic Rules:**
Provide exactly 5-7 rules in the following format:
1. [Rule description]
2. [Rule description]
...
```

Figure 3: Base meta-prompt before refinement. This prompt lacked systematic analysis steps and explicit validation criteria, leading to less effective rule extraction.

Stage 1: Rubric Generation and Critique Prompt

```
You are an expert prompt engineer and AI systems analyst. Your task is to perform a brutal,
honest critique of a prompt designed to extract heuristic rules to improve model performance.

### TASK
The base prompt is designed to analyze evaluation results and extract heuristic rules to
improve LLM performance on the following task:

{task_instructions}

### BASELINE METRICS
Current performance on this task:
- **Action Pass@1**: {action_pass_rate}% - Model selects the correct tool
- **Parameter Pass@1**: {param_pass_rate}% - Model populates correct parameter values
- **E2E Pass@1**: {e2e_pass_rate}% - Both action and parameters correct
- **Total Examples**: {total}

### PROBLEM
Rules extracted by the base prompt either don't improve these metrics or actively decrease
performance.

### BASE PROMPT TO CRITIQUE
{base_prompt}

The base prompt must work across different API domains (e.g., BMR/TMR, Finance, Search).
Example API:

{api_docs}

### EXAMPLE FAILURE CASE
Below is one failure example to help you assess if the base prompt's analysis approach would
work:

{failure_example}

### YOUR TASK
1. Create a critique rubric that evaluates:
   **STRUCTURAL REQUIREMENTS (Must-Have):**
   a) Role & Task Definition - Does it clearly state the role? Does it require analyzing BOTH
successes and failures? Does it explain the task context (what model was doing, with what API)
BEFORE presenting baseline metrics? Are actionable heuristics explicitly required?
   b) Problem Statement - Does it provide clear placeholders/sections for: (i) the system
instruction given to the model, (ii) the API documentation the model used, (iii) baseline
zero-shot performance metrics? Are these properly labeled and positioned after task context?
   c) Optimization Goals - Does it specify: (i) improve performance metrics, (ii) reduce token
generation for lower latency?
   d) Data Structure - Does it explain the example format (input, preferred_output,
non_preferred_output)?
   e) Example Walkthrough - Does it demonstrate analysis on one example following its analysis
steps?
   f) Analysis Steps - Are the detailed analysis steps clear, systematic, and effective?
   g) Examples Placeholder - Is there a clear section to provide all examples?
   h) Analysis Execution - Does it guide how to conduct the analysis across all examples?
   i) Output Format - Does it specify a structured output format (e.g., JSON)? Is the schema
clearly defined with required fields? Is it parseable by standard tools? Does it include all
necessary information for extracting actionable heuristics (condition, action, examples, test
criteria)?

   **ANALYSIS METHODOLOGY (Your Focus):**
   Design 5-8 additional criteria specifically evaluating the analysis steps (item f above).
   Consider: Does it analyze successes vs only failures? Root cause identification?
Generalizability? Actionability?
```

Figure 4: Stage 1: Rubric Generation and Critique Prompt. This prompt guides the LRM to generate a comprehensive rubric and critique the base meta-prompt.

Stage 1: Rubric Generation and Critique Prompt (Cont.)

```
2. Analyze the base prompt against your complete rubric
3. For EACH rubric criterion (a-i and any additional), provide:
   - Detailed critique with specific examples
   - Letter grade (A+, A, A-, B+, B, B-, C+, C, C-, D+, D, D-, F)
   - Reason for the grade
4. Provide an overall letter grade based on all criteria

### OUTPUT FORMAT
You MUST respond with ONLY a valid JSON object with exactly these keys:

```json
{
  "rubric": "Your critique rubric with all criteria",
  "critiques": {
    "criterion_1_key": {"grade": "[letter grade]", "reason": "[why this grade]", "details":
"[specific examples and analysis]"},
    "criterion_2_key": {"grade": "[letter grade]", "reason": "[why this grade]", "details":
"[specific examples and analysis]"},
    "criterion_N_key": {"grade": "[letter grade]", "reason": "[why this grade]", "details":
"[specific examples and analysis]"}
  },
  "overall_grade": "[letter grade]",
  "overall_summary": "Brief summary of strengths and critical weaknesses"
}
```

Note: Use descriptive keys for each criterion (e.g., 'role_task_definition',
'analysis_methodology', etc.). Include all criteria from the structural requirements (a-i) plus
any additional criteria you design.

Be brutally honest. If the prompt is fundamentally flawed, say so explicitly.
```

Figure 5: Stage 1: Rubric Generation and Critique Prompt (Cont.). This prompt guides the LRM to generate a comprehensive rubric and critique the base meta-prompt.

Stage 2: Refinement Prompt

```
You are an expert prompt engineer. Your task is to generate an improved prompt for extracting
effective heuristic reasoning rules from model failures.

### CONTEXT
The goal is to create a prompt that:
- Analyzes LLM failures on tool-use tasks
- Extracts actionable, general-purpose heuristics
- Generates rules that ACTUALLY improve performance when appended to task prompts

### CRITIQUE OF PREVIOUS PROMPT
{critique_analysis}

### CRITIQUE RUBRIC
{critique_rubric}

### EXAMPLE TASK CONTEXT
Below is ONE example of the task domain. Your prompt must work across different APIs/tools.

Example Task Instructions: {task_instructions}

Example API Documentation: {api_docs}
```

Figure 6: Stage 2: Refinement prompt. This guides the LRM to generate improved prompts addressing identified weaknesses.

Stage 2: Refinement Prompt (Cont.)

```
### TASK DOMAIN
This prompt will be used to extract heuristics from LLM failures on tool-use tasks.
The task involves (across DIFFERENT APIs/tools):
- Selecting the correct API/tool from documentation
- Populating parameters with correct values from user queries
- Handling conversation history and context

NOTE: The example below is just ONE instance. Your prompt must generalize to other APIs/tools.

### YOUR TASK
Generate an improved prompt that addresses ALL the flaws identified in the critique.

### REQUIREMENTS
1. The prompt must guide the LLM to extract heuristics that:
   - Are GENERAL-PURPOSE and transferable across ANY API/tool domain
   - Anticipate common failure modes (action selection, parameter population, context
     handling)
   - Are actionable and specific enough to change behavior
   - Avoid overfitting to the specific example API shown above

2. Output format:
   - No special tokens (no <thinking>, <output>, etc.)
   - Use simple delimiters (---, ###)
   - Rules should be numbered and easily extractable

3. The prompt should:
   - Provide clear examples of good vs bad rules
   - Guide deep analysis of failure patterns
   - Encourage structural/systematic thinking
   - Prevent overfitting to specific examples

### OUTPUT FORMAT
Provide your improved prompt as a JSON-compatible array of strings.
Each string is one line of the prompt.

--- IMPROVED PROMPT ---
[Your improved prompt here, formatted as JSON array]
```

Figure 7: Stage 2: Refinement prompt (Cont.). This guides the LRM to generate improved prompts addressing identified weaknesses.

Stage 2: Self-Critique Prompt

```
You are a harsh, uncompromising prompt evaluation expert. Your task is to critique a prompt
designed to extract heuristic reasoning rules.

### CRITIQUE RUBRIC
{critique_rubric}

### PROMPT TO EVALUATE
{generated_prompt}

### YOUR TASK
Evaluate the prompt against EACH criterion in the rubric.

For each criterion:
1. Score it (Excellent / Good / Adequate / Poor / Failing)
2. Provide specific evidence from the prompt
3. Identify concrete weaknesses
```

Figure 8: Stage 2: Self-critique prompt. This evaluates generated prompts against the fixed rubric.

Stage 2: Self-Critique Prompt (Cont.)

```
### OUTPUT FORMAT

--- CRITERION-BY-CRITERION EVALUATION ---
[For each criterion, provide assessment]

--- OVERALL ASSESSMENT ---
Strengths:
- [List 2-3 key strengths]

Critical Weaknesses:
- [List all significant flaws]

Grade: [A+, A, A-, B+, B, B-, C+, C, C-, D, or F]

Grade Justification:
[Explain why this specific grade, not higher or lower]

--- WHAT PREVENTS A+ GRADE ---
[If grade is not A+, be BRUTALLY specific about what is missing or wrong]
[List concrete, actionable gaps that prevent the highest grade]

Be harsh. Grade inflation helps no one. If it's not truly excellent, don't give it an A+.
```

Figure 9: Stage 2: Self-critique prompt (Cont.). This evaluates generated prompts against the fixed rubric.

Meta-Prompt $\mathcal{P}^*_{\text{meta}}$

```
You are an expert at analyzing LLM tool-use behavior (both successes and failures) and
extracting actionable reasoning rules.

**YOUR TASK:**
Analyze model behavior on tool-use tasks to extract heuristic rules that improve performance.
You will examine both successful actions (preferred_output) and failures (non_preferred_output)
to identify patterns.

**TASK CONTEXT:**
The model was evaluated on tool-use tasks where it must:
1. Select the correct API function to call
2. Populate parameters with correct values from user input
3. Handle API errors appropriately

**COMMON API DOCUMENTATION:**
All examples share the same API documentation:
{api_documentation}

**BASELINE SYSTEM INSTRUCTION:**
The model was given this system instruction during evaluation:
{system_instruction}

**BASELINE PERFORMANCE:**
- Action Pass@1: {action_pass_rate}% - Model selected correct function
- Parameter Pass@1: {param_pass_rate}% - Model populated parameters with correct values
- E2E Pass@1: {e2e_pass_rate}% - Both action AND parameters correct
- Total examples: {total}

**OPTIMIZATION GOALS:**
1. **Improve accuracy**: Increase Action Pass@1, Parameter Pass@1, and E2E Pass@1 rates shown
  above
2. **Reduce latency**: Extract concise, actionable rules that minimize token generation when
  added to system prompts

**DATA STRUCTURE:**
Each example contains:
- `input`: Full conversation history (user queries and model responses)
- `preferred_output`: Correct next action (what model SHOULD have done)
- `non_preferred_output`: Incorrect action (what model ACTUALLY did)

**IMPORTANT:** All examples use the common API documentation shown above. When analyzing
  failures, check if:
- The API docs clearly specified the correct format/values
- The model misread or ignored the documentation
- The documentation was ambiguous or misleading

**EXAMPLE WALKTHROUGH (to show expected analysis depth):**

```
Common API docs specify: 'inimperial (string, REQUIRED): ...' with no example values shown
Example input: 'Calculate BMR for 30-year-old female, 65kg, 170cm'

Model did: bmr_index_for_bmr_and_tmr({{'inimperial': 'metric', 'sex': 'female', 'age': 30,
'weight': 65, 'height': 170}})
Should have done: bmr_index_for_bmr_and_tmr({{'inimperial': 'false', 'sex': 'female', 'age':
30, 'weight': 65, 'height': 170}})

Analysis:
- Failure type: Parameter format error
- Specific issue: Used 'metric' instead of 'false' for inimperial parameter
```

Figure 10: Complete meta-prompt $\mathcal{P}^*_{\text{meta}}$ for rule extraction (Part 1/9).
```

Meta-Prompt $\mathcal{P}^*_{\text{meta}}$ (Cont.)

```
- Root cause: Misinterpreted parameter name as accepting unit system ('metric'/'imperial'). API
  docs didn't show example values.
- Actual API expectation: String 'true' or 'false' indicating whether to use imperial units
- Documentation issue: Parameter name is misleading (suggests 'imperial'/'metric' values)
- Frequency: If this pattern appears in 8+ examples, it needs a rule
- Rule needed: 'When parameter name suggests boolean or binary choice, check API docs for
  example values. If no examples shown, test with common boolean strings ("true"/"false")
  rather than synonyms. Never infer format from parameter name alone.'
```

Figure 11: Complete meta-prompt $\mathcal{P}^*_{\text{meta}}$ for rule extraction (Part 2/9).

Meta-Prompt $\mathcal{P}^*_{\text{meta}}$ (Cont.)

```
## ANALYSIS EXECUTION INSTRUCTIONS

You will analyze ALL {total} examples provided below. For each example:
1. Compare non_preferred_output (what model did) vs preferred_output (what model should have
  done)
2. Identify the failure type(s) and categorize according to Step 1 categories
3. Note specific details (parameter names, values, error messages) for Steps 2-3
4. Track which examples exhibit each pattern to ensure accurate frequency counts

Work systematically through all examples before proceeding to extract rules. Ensure every
example is accounted for in your analysis.

---

## STEP 1: FAILURE FREQUENCY ANALYSIS

Count occurrences of each failure type across ALL {total} examples. Format: 'Failure type: X
examples'

**ACTION SELECTION FAILURES:**
- Wrong function called when correct one available: ___ examples
- Repeated same failed function call: ___ examples
- Called Finish prematurely (before gathering all data): ___ examples
- No alternative tried after API failure: ___ examples
- Called wrong function after API error: ___ examples

**PARAMETER POPULATION FAILURES:**
- Wrong format (e.g., 'metric' vs 'false', boolean vs string): ___ examples
- Failed to extract user-provided values: ___ examples
- Used placeholder/example values instead of actual data: ___ examples
- Omitted required parameters: ___ examples
- Wrong data type (string vs number, etc.): ___ examples
- Incorrect parameter value despite correct format: ___ examples
- Misread API documentation (docs were clear but model ignored): ___ examples
- Ambiguous API docs led to wrong choice: ___ examples

**API ERROR HANDLING FAILURES:**
- Didn't detect HTML error page (service down): ___ examples
- Didn't handle 503/timeout errors: ___ examples
- Didn't recognize empty/null response as error: ___ examples
- Retried failed API too many times: ___ examples

---
```

Figure 12: Complete meta-prompt $\mathcal{P}^*_{\text{meta}}$ for rule extraction (Part 3/9).

Meta-Prompt $\mathcal{P}^*_{\text{meta}}$ (Cont.)

```
## STEP 2: PARAMETER FORMAT DEEP ANALYSIS

For each parameter format mismatch found, document:

**Parameter: [name]**
- Expected format: [what API docs specified]
- What model used: [actual value/format used]
- Frequency: ___ examples
- What API docs said: [exact specification from docs]
- Root cause: [why model made this mistake]

Example:
**Parameter: inimperial**
- Expected format: string 'true'/'false'
- What model used: 'metric' or boolean
- Frequency: 8 examples
- What API docs said: 'string, REQUIRED' (no examples)
- Root cause: inferred from param name

(Repeat for all parameter format issues you find.)

---

## STEP 3: API ERROR PATTERN ANALYSIS

For each error type found, document:

**Error Pattern 1: HTML Error Page**
- Example response: '<!DOCTYPE html>...Application Error...'
- Frequency: ___ examples
- What model currently does: [describe behavior]
- What model should do: Detect HTML response (starts with '<'), recognize service is down, try
  alternative function once or call Finish with explanation
- Detection rule: 'If response is string starting with "<!DOCTYPE" or "<html>", it's service
  error'

**Error Pattern 2: 503/Timeout**
- Example response: 'status_code=503' or 'Read timed out'
- Frequency: ___ examples
- What model currently does: [describe behavior]
- What model should do: Retry same call once, then try alternative function, then give up
- Detection rule: 'If error contains "503" or "timeout" or "timed out", it's temporary failure'

**Error Pattern 3: Empty Response**
- Example response: '{{{{}}}}' or '[]' or ''
- Frequency: ___ examples
- What model currently does: [describe behavior]
- What model should do: Check if empty is valid (e.g., no results found) vs error, handle
  accordingly
- Detection rule: 'If response is empty dict/list/string, check if this is expected result or
  error'

(Add all error patterns you find)

---
```

Figure 13: Complete meta-prompt $\mathcal{P}^*_{\text{meta}}$ for rule extraction (Part 4/9).

Meta-Prompt $\mathcal{P}^*_{\text{meta}}$ (Cont.)

```
## STEP 4: ROOT CAUSE IDENTIFICATION

For the 5 most frequent failures, identify root cause:

**Failure 1:** [Most frequent - describe symptom]
- Frequency: ___ examples
- Symptom: [what we observe]
- Root Cause: [why it happens - be specific]
- Fix: [what rule would prevent it]
- Impact: [how much would fixing this improve pass rate]

**Failure 2:** [Second most frequent]
- Frequency: ___ examples
- Symptom: [what we observe]
- Root Cause: [why it happens]
- Fix: [what rule would prevent it]
- Impact: [estimated improvement]

(Continue for top 5)

---

## STEP 5: SUCCESS PATTERN ANALYSIS

For each failure pattern identified in Steps 1-4, analyze preferred_output examples to find
mitigation strategies:

**For each top failure pattern:**

**Failure Pattern:** [From Step 1, e.g., 'Wrong parameter format - 8 examples']
- **What failed examples did wrong:** [Describe the incorrect behavior]
- **What successful examples did right:** [Analyze preferred_output for similar scenarios]
- **Key difference:** [What distinguishes success from failure]
- **Mitigation strategy:** [Actionable rule derived from successful behavior]

Repeat for each major failure pattern. Focus on extracting concrete strategies from
preferred_output that would prevent the failures.

---
```

Figure 14: Complete meta-prompt $\mathcal{P}^*_{\text{meta}}$ for rule extraction (Part 5/9).

Meta-Prompt $\mathcal{P}^*_{\text{meta}}$ (Cont.)

```
## STEP 6: EXTRACT RULES (PRIORITIZED BY IMPACT)

Create rules covering all failure patterns occurring in 3+ examples, using mitigation
strategies from Step 5.
Typical output: 8-15 rules (more if needed to cover all patterns).

Output rules as a JSON array. Each rule MUST include all these fields:

```json
[
  {{
    "rule_id": 1,
    "name": "Short Descriptive Name",
    "priority": "High|Medium|Low",
    "category": "Action Selection|Parameter Population|Error Handling",
    "addresses": "Specific failure from Step 1, with frequency",
    "condition": "Specific, testable WHEN condition",
    "action": "Specific, implementable THEN action",
    "example_wrong": "Actual failed behavior from examples",
    "example_correct": "What should have happened",
    "test_criterion": "How to verify this rule is being followed",
    "estimated_impact": "Percentage of failures this would prevent"
  }}
]
```

RULE QUALITY REQUIREMENTS:
- [X] Addresses failure occurring in 3+ examples
- [X] Includes specific WHEN condition (not vague)
- [X] Includes specific THEN action (directly implementable)
- [X] References actual examples from the data
- [X] Has clear test criterion
- [X] Doesn't conflict with other rules
- [X] Is generalizable across different APIs/domains
- [X] Is concise and doesn't add unnecessary verbosity to prompts

GOOD RULE EXAMPLE:

```json
[
  {{
    "rule_id": 1,
    "name": "Boolean-String Parameter Format",
    "priority": "High",
    "category": "Parameter Population",
    "addresses": "Parameter format errors (8 examples)",
    "condition": "When parameter name contains boolean indicators (is/has/enable/disable) OR
suggests binary choice (imperial/metric)",
    "action": "Check API documentation for exact string format. If docs show string examples
like 'true'/'false', use those exact strings. Never use synonyms (e.g., 'metric' for 'false')
or boolean types.",
    "example_wrong": "{{'inimperial': 'metric'}} or {{'inimperial': false}}",
    "example_correct": "{{'inimperial': 'false'}}",
    "test_criterion": "Check if parameter value matches exact string from API docs",
    "estimated_impact": "8% of parameter failures"
  }},
  {{
    "rule_id": 2,
    "name": "HTML Error Detection",
    "priority": "High",
```

Figure 15: Complete meta-prompt $\mathcal{P}^*_{\text{meta}}$ for rule extraction (Part 6/9).
```

Meta-Prompt $\mathcal{P}^*_{\text{meta}}$ (Cont.)

```
    "category": "Error Handling",
    "addresses": "Repeated failed API calls (12 examples)",
    "condition": "When API response is a string starting with '<!DOCTYPE' or '<html>' or
contains 'Application Error'",
    "action": "Recognize this as service unavailability (not parameter error). Try alternative
function once. If no alternative exists, call Finish with explanation. Never retry same
function.",
    "example_wrong": "Retried bmr_index 3 times with same params after HTML error",
    "example_correct": "After HTML error, tried tmr_index instead, then called Finish",
    "test_criterion": "Check if model stops after 1 HTML error and tries alternative",
    "estimated_impact": "12% of action failures"
  }}
]
```

Figure 16: Complete meta-prompt $\mathcal{P}^*_{\text{meta}}$ for rule extraction (Part 7/9).

Meta-Prompt $\mathcal{P}^*_{\text{meta}}$ (Cont.)

```
## STEP 7: VALIDATE RULES

For each rule you created, verify:

1. **Coverage check:** Does it prevent the failure in 3+ examples? (Yes/No + count)
2. **Regression check:** Could this rule cause currently correct predictions to become
  incorrect? Check if applying this rule to preferred_output examples would break them. (Yes/No
  + examples if yes)
3. **Atomicity check:** Does this rule address exactly ONE failure pattern, or does it try to
  cover multiple unrelated issues? (Atomic/Not atomic)
4. **Implementability:** Can it be added to a prompt as a single clear instruction? (Yes/No)
5. **Testability:** Is the test criterion measurable and unambiguous? (Yes/No)
6. **Conflict check:** Does it conflict with any other rule? (Yes/No + which rule)
7. **Generalizability:** Does it apply across different APIs, not just one? (Yes/No)

If any rule fails validation, revise or remove it. Split non-atomic rules into separate rules.

---

## STEP 8: META-ANALYSIS

**Coverage Analysis:**
- Total failures in dataset: {total} × (1 - {e2e_pass_rate}/100) = ___ failures
- Failures addressed by rules: ___ (sum of estimated impacts)
- Coverage: ___% of failures would be prevented
- Remaining gaps: [List failure types NOT covered by rules and why]

**Expected Improvement:**
- Current Action Pass@1: {action_pass_rate}%
- Estimated new Action Pass@1: ___% (based on action rules)
- Current Parameter Pass@1: {param_pass_rate}%
- Estimated new Parameter Pass@1: ___% (based on parameter rules)
- Current E2E Pass@1: {e2e_pass_rate}%
- Estimated new E2E Pass@1: ___% (based on all rules)

**Uncovered Patterns:**
List any failure patterns that occur in <3 examples (too rare for rules):
- [Pattern 1]: ___ examples - [why no rule needed]
- [Pattern 2]: ___ examples - [why no rule needed]

---

## EXAMPLES TO ANALYZE

{examples}

---
```

Figure 17: Complete meta-prompt $\mathcal{P}^*_{\text{meta}}$ for rule extraction (Part 8/9).

Meta-Prompt $\mathcal{P}^*_{\text{meta}}$ (Cont.)

```
## FINAL OUTPUT FORMAT

Provide your complete analysis following Steps 1-8 above, then conclude with:

**FINAL EXTRACTED RULES:**

Output as JSON array (ordered by priority):
```json
[
  {{ rule objects from Step 6 }}
]
```

VALIDATION SUMMARY:

For each rule, provide validation results from Step 7 criteria.

META-ANALYSIS SUMMARY:

Provide coverage analysis and expected improvement metrics from Step 8.

IMPLEMENTATION RECOMMENDATIONS

Provide deployment strategy as JSON:
```json
{{
  "critical_priority": {{
    "rule_ids": [1, 2, 3],
    "rationale": "Why these rules should be deployed first",
    "estimated_impact": "X% improvement in E2E Pass@1"
  }},
  "high_priority": {{
    "rule_ids": [4, 5, 6],
    "rationale": "Why these rules should be deployed second",
    "estimated_impact": "X% additional improvement"
  }},
  "medium_priority": {{
    "rule_ids": [7, 8, 9],
    "rationale": "Why these rules address edge cases",
    "estimated_impact": "X% additional improvement"
  }}
}}
```
```

Figure 18: Complete meta-prompt $\mathcal{P}^*_{\text{meta}}$ for rule extraction (Part 9/9).

Example Rule in JSON Format

```
{
  "rule_id": 1,
  "name": "HTML Error Detection",
  "priority": "High",
  "category": "Error Handling",
  "addresses": "Repeated failed API calls after service errors (12 examples)",
  "condition": "When API response is a string starting with '<!DOCTYPE' or
               '<html>' or contains 'Application Error'",
  "action": "Immediately recognize this as service unavailability (NOT a
            parameter error). Try alternative function once if available.
            If no alternative or alternative fails, call Finish with
            explanation. Never retry same function after HTML error.",
  "example_wrong": "Received HTML error from bmr_index_for_bmr_and_tmr,
                   then retried bmr_index_for_bmr_and_tmr with same
                   parameters 3 times",
  "example_correct": "Received HTML error from bmr_index_for_bmr_and_tmr,
                     tried tmr_index_for_bmr_and_tmr once, then called
                     Finish with explanation",
  "test_criterion": "Check if model stops retrying after detecting HTML
                    error and either tries alternative once or calls Finish",
  "estimated_impact": "12% of action selection failures"
}
```

Figure 19: Example extracted rule in JSON format showing all required fields. This rule addresses the failure mode of repeatedly retrying APIs after service-level errors.

Example Rule in Serialized Natural Language Format

```
Rule 1:
  WHEN: When API response is a string starting with '<!DOCTYPE' or '<html>'
        or contains 'Application Error'
  THEN: Immediately recognize this as service unavailability (NOT a parameter
        error). Try alternative function once if available. If no alternative
        or alternative fails, call Finish with explanation. Never retry same
        function after HTML error.
  WRONG BEHAVIOR: Received HTML error from bmr_index_for_bmr_and_tmr, then
                  retried bmr_index_for_bmr_and_tmr with same parameters 3
                  times
  CORRECT BEHAVIOR: Received HTML error from bmr_index_for_bmr_and_tmr, tried
                    tmr_index_for_bmr_and_tmr once, then called Finish with
                    explanation
  TEST CRITERION: Check if model stops retrying after detecting HTML error and
                  either tries alternative once or calls Finish
```

Figure 20: Serialized natural language version of the rule from Figure 19. This format is inserted into inference prompts.

LLM-as-Judge Filtering Prompt

```
You are an expert evaluator for AI agent action selection tasks. Your job is to identify the
BEST action between two options given the context.

# Task
Given a user query, conversation history, available APIs, and two possible actions (Action A
and Action B), determine which action is BEST and whether it's significantly better.

# Evaluation Criteria
Compare both actions on:
1. **Correctness**: Does it match what the user requested?
2. **Parameter Quality**: Are parameters accurate, complete, and optimal?
3. **Logical Next Step**: Is it the most logical and appropriate next step given the
  conversation history?
4. **Context Awareness**: Does it consider conversation history and previous attempts?
5. **API Usage**: Does it use the most appropriate API/tool?

# Decision Rules
- Mark as VALID only if one action is CLEARLY BETTER than the other AND is the most logical
  next step
- If both actions are equally good or bad, mark as INVALID
- If both would fail, mark as INVALID
- If neither action is the most logical next step based on conversation history, mark as
  INVALID (even if one is technically correct)
- The best action must be: (1) correct, (2) superior to the alternative, AND (3) the most
  logical next step given the context

# Examples

## Example 1: VALID (Action A is best)
User requested: Get social sentiment data surrounding specific stocks on Twitter
Action A: Call API with social parameter set to 'twitter'
Action B: Call API with social parameter set to 'stocktwits'
Verdict: VALID - Action A directly matches user's explicit request for Twitter data

## Example 2: INVALID (Both reasonable)
User requested: Search for phones
Previous attempts: None
Action A: Search for 'phone'
Action B: Search for 'smartphone'
Verdict: INVALID - Both are reasonable first attempts, no clear winner

## Example 3: VALID (Action B is best)
User requested: Search for phones
Previous attempts: Searched 'phone' twice, got same results
Action A: Search for 'phone' again
Action B: Search for 'smartphone' instead
Verdict: VALID - Action B adapts to context, Action A repeats failed attempt

## Example 4: INVALID (Neither is logical next step)
User requested: Find product reviews
Previous attempts: None, just started conversation
Action A: Delete user account
Action B: Update payment method
Verdict: INVALID - Both are valid actions but neither is the logical next step for finding
reviews
```

Figure 21: Qwen3-30B prompt used for filtering preference pairs. The prompt evaluates whether candidate actions represent optimal next steps given the state.

LLM-as-Judge Filtering Prompt (Cont.)

```
# Input Format
INSTRUCTION:
{instruction_data}

CONVERSATION HISTORY:
{input_data}

ACTION A:
{action_a}

ACTION B:
{action_b}

# Your Task
Analyze the input and respond with a JSON object:
{
  "verdict": "VALID" or "INVALID",
  "best_action": "A" or "B" or "NONE",
  "reasoning": "Brief explanation comparing both actions and why one is clearly better",
  "confidence": "high" or "medium" or "low"
}

Note: Set best_action to "NONE" if verdict is INVALID
```

Figure 22: Qwen3-30B prompt used for filtering preference pairs (Cont.). The prompt evaluates whether candidate actions represent optimal next steps given the state.

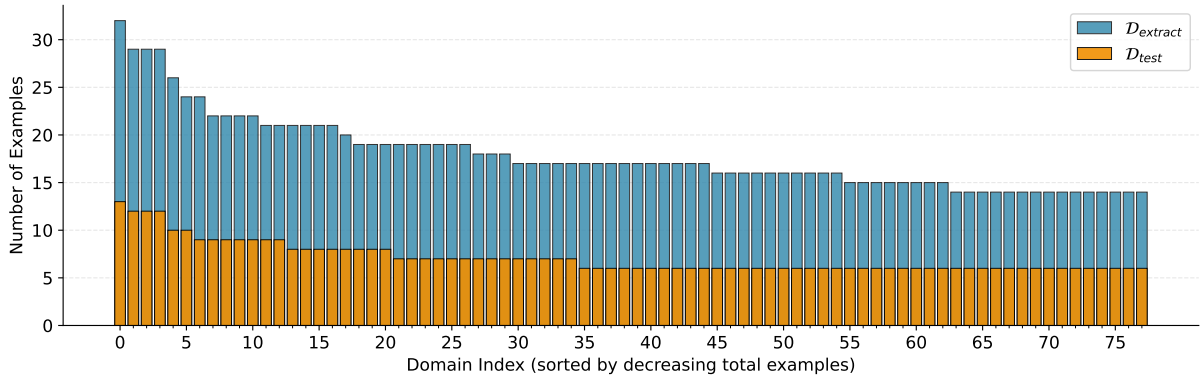


Figure 23: Distribution of examples across 78 domains in $\mathcal{D}_{\text{extract}}$ and $\mathcal{D}_{\text{test}}$. Domains are indexed in decreasing order of total examples. Each domain defines a unique API documentation $\mathcal{D}_{\text{API}}$ with corresponding action space $\mathcal{A}$.

Zero-Shot Baseline Prompt $\mathcal{P}_{\text{zero-shot}}$

```
{instruction_data}

{api_documentation}

--- CONVERSATION HISTORY ---
{input_data}

--- TASK ---
Based on the instructions, API documentation, and conversation history, what is the next
thought and action?

Assistant:
```

Figure 24: Zero-shot baseline prompt template.

Zero-Shot Baseline Prompt $\mathcal{P}_{\text{zero-shot-cot}}$

```
{instruction_data}

{api_documentation}

--- CONVERSATION HISTORY ---
{input_data}

--- TASK ---
Think step by step, examine previous steps and thoughts, then decide: Based on the instructions,
API documentation, and conversation history, what is the next thought and action?

Assistant:
```

Figure 25: Zero-shot-CoT baseline prompt structure with explicit reasoning instruction.

Few-Shot Baseline Prompt $\mathcal{P}_{\text{few-shot}}$

```
{instruction_data}

{api_documentation}

Here are some examples of how to perform the task:
{few_shot_examples}
--- END OF EXAMPLES ---

--- CONVERSATION HISTORY ---
{input_data}

--- TASK ---
Based on the instructions, API documentation, and conversation history, what is the next
thought and action?

Assistant:
```

Figure 26: Few-shot baseline prompt structure. The same template is used for 1/2/3-shot variants.

GEAR Inference Prompt Template $\mathcal{P}_{\mathbf{GEAR}}$

```
{instruction_data}

{api_documentation}

{guidelines}

--- CONVERSATION HISTORY ---
{input_data}

--- TASK ---
Based on the instructions, API documentation, and conversation history, what is the next
thought and action?

Assistant:
```

Figure 27: **GEAR** inference prompt template. Rules are inserted between API documentation and conversation history, providing explicit behavioral constraints during action selection.

GEAR+2-shot Inference Prompt Template $\mathcal{P}_{\text{GEAR+2-shot}}$

```
{instruction_data}

{api_documentation}

{guidelines}

Here are some examples of how to perform the task:
{few_shot_examples}
--- END OF EXAMPLES ---

--- CONVERSATION HISTORY ---
{input_data}

--- TASK ---
Based on the instructions, API documentation, and conversation history, what is the next
thought and action?

Assistant:
```

Figure 28: **GEAR**+2-shot inference prompt template. Examples placed between rules and query.

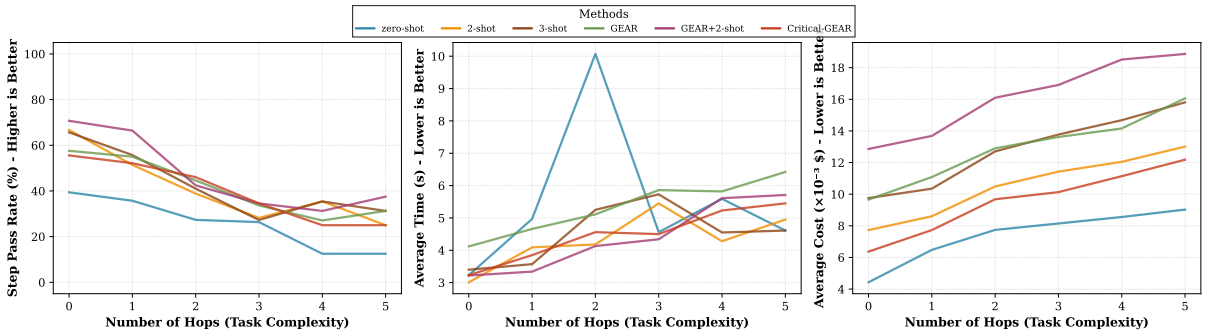


Figure 29: Performance, latency, and cost as a function of task complexity (number of hops) on Qwen3-30B. All methods degrade with complexity, but GEAR+2-shot maintains highest performance across all levels while achieving competitive latency and cost.

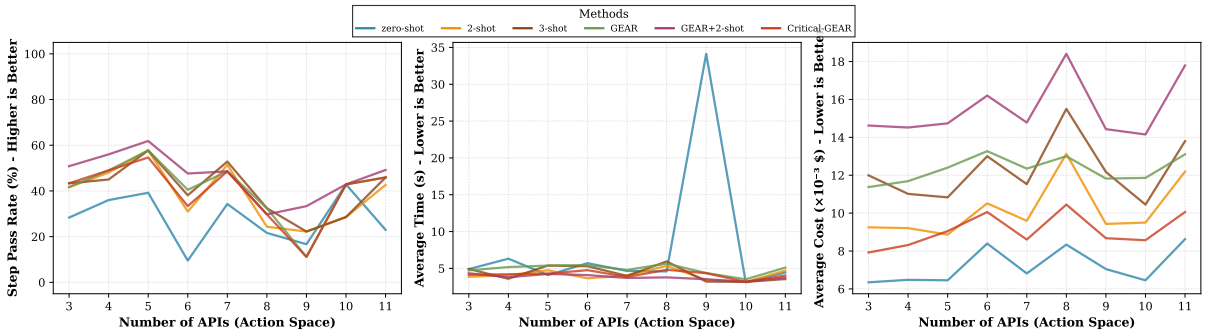


Figure 30: Performance, latency, and cost as a function of action space complexity (number of APIs) on Qwen3-30B. Unlike trajectory complexity, performance does not show clear degradation with larger action spaces.

Method	Pass@1 (%)			Gain (%)	Efficiency			
	Action	Parameter	Step	ΔStep (%)	Latency (s)	ΔLat. (%)	In Tok.	Out Tok.
Qwen3-30B-A3B								
Zero-shot	57.53	42.72	30.20	–	**0.63**	–	1,299	83
Zero-shot-CoT	57.94	41.35	30.68	+1.6	0.64	−1.6	1,313	89
1-shot	64.64	51.36	38.84	+28.6	0.68	−7.9	1,823	89
2-shot	66.43	54.05	43.11	+42.7	0.69	−9.5	2,366	89
3-shot	70.97	58.00	47.78	+58.2	0.69	−9.5	2,961	90
GEAR (ours)	72.81	56.25	47.74	+58.1	0.85	−34.9	2,850	114
Critical-**GEAR** (ours)	65.25	49.91	38.40	+27.2	0.72	−14.3	1,858	101
GEAR+2-shot (ours)	**82.25**⋆	**65.24**⋆	**59.69**⋆	+97.6	0.78	−23.8	4,446	101
2-shot+**GEAR** (ours)	78.68	63.85	57.79	+91.4	0.80	−27.0	4,469	106
DirectiveOnly-**GEAR**	65.00	51.38	40.97	+35.7	0.83	−31.7	2,104	116
PedagogyOnly-**GEAR**	66.35	51.40	41.61	+37.8	0.75	−19.0	2,070	105
GEAR+1-shot	70.74	55.39	46.44	+53.8	0.84	−33.3	3,370	114
GEAR+3-shot	71.99	57.49	48.48	+60.5	0.83	−31.7	4,508	109
RAG-style 3-shot (query only)	60.01	46.60	34.00	+12.6	0.71	−12.7	2,975	94
RAG-style 3-shot (query+history)	64.06	50.53	39.30	+30.1	0.72	−14.3	3,165	94
Qwen3-4B								
Zero-shot	57.53	37.57	26.45	–	**0.80**	–	1,299	84
Zero-shot-CoT	57.57	39.28	28.07	+6.1	0.86	−7.5	1,313	92
1-shot	62.88	48.93	37.78	+42.8	0.89	−11.2	1,823	93
2-shot	63.72	51.21	39.79	+50.4	0.94	−17.5	2,366	97
3-shot	68.09	57.41	45.15	+70.7	1.02	−27.5	2,961	104
GEAR (ours)	72.51	53.55	45.48	+71.9	1.17	−46.2	2,854	121
Critical-**GEAR** (ours)	63.42	45.17	33.96	+28.4	0.99	−23.8	1,830	104
GEAR+2-shot (ours)	**76.72**	**62.97**	**56.29**	+112.8	1.13	−41.2	4,268	108
2-shot+**GEAR** (ours)	72.22	58.07	47.48	+79.5	1.32	−65.0	3,917	132
DirectiveOnly-**GEAR**	66.42	47.01	36.59	+38.3	1.16	−45.0	2,103	122
PedagogyOnly-**GEAR**	64.28	49.25	38.57	+45.8	1.07	−33.8	2,077	112
GEAR+1-shot	69.50	53.94	44.59	+68.6	1.65	−106.2	3,375	121
GEAR+3-shot	71.96	59.16	48.68	+84.0	1.71	−113.8	4,505	122
RAG-style 3-shot (query only)	58.55	46.90	32.12	+21.4	1.51	−88.8	2,976	108
RAG-style 3-shot (query+history)	59.74	46.30	32.84	+24.2	1.56	−95.0	3,165	110
Ministral3-14B								
Zero-shot	53.25	36.06	24.42	–	1.44	–	1,343	147
Zero-shot-CoT	53.76	37.32	25.69	+5.2	1.51	−4.9	1,357	158
1-shot	58.25	43.15	32.44	+32.8	**1.41**	+2.1	1,898	143
2-shot	61.90	47.67	37.14	+52.1	1.40	+2.8	2,472	140
3-shot	62.93	48.28	38.42	+57.3	1.43	+0.7	3,101	142
GEAR (ours)	**70.19**	49.49	**42.53**	+74.2	1.61	−11.8	3,008	159
Critical-**GEAR** (ours)	60.27	44.00	33.11	+35.6	1.53	−6.2	1,945	153
GEAR+2-shot (ours)	67.74	46.71	38.49	+57.6	1.67	−16.0	4,131	162
2-shot+**GEAR** (ours)	68.91	**51.44**	41.35	+69.3	1.67	−16.0	4,131	161
DirectiveOnly-**GEAR**	63.22	44.27	35.26	+44.4	1.62	−12.5	2,191	163
PedagogyOnly-**GEAR**	63.66	44.86	35.80	+46.6	1.49	−3.5	2,189	148

Table 15: **GEAR** results on $\mathcal{D}_{\text{test}}$ across open-model configurations. Step Pass@1 (%) is the primary metric. ΔStep / ΔLat. = relative change vs each model's own Zero-shot (positive ΔLat. = faster). Best results per model in **bold**; ⋆overall best across configurations. Row colors: **GEAR** variants; ablation studies.