CROW: CONSTRAINT-GUIDED AGENTIC WORKFLOWS FOR MATHEMATICAL REASONING

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models have demonstrated strong performance on mathematical reasoning tasks, yet reasoning trajectories that violate critical conditions required for a valid solution can be difficult to correct. Existing approaches primarily focus on improving reasoning generation, but provide limited mechanisms for enforcing the constraints that should govern the reasoning process. We argue that critical reasoning constraints provide explicit conditions for guiding and verifying reasoning, particularly on challenging mathematical problems. We introduce **C**onstraint **R**euse and **O**rchestration **W**orkflow (CROW), a constraint-guided agentic workflow for mathematical reasoning. CROW formulates constraint utilization as a sequential decision-making problem, allowing the orchestrator to selectively retrieve, inspect, reuse, disregard, or propose constraints according to the current problem. The orchestration policy is optimized with reinforcement learning based on downstream solution correctness. During training, newly proposed constraints that contribute to successful solutions are retained for future use, progressively expanding the set of available constraints. Through reinforcement learning (RL), the orchestrator learns when to acquire and apply the retrieved constraint, and when to disregard it due to limited relevance or applicability to the current problem. Experiments on HARDMath, HARDMath2, and DeepMath show that CROW substantially improves performance on HARDMath, where critical constraints contribute to the largest gains, while RL-trained orchestration preserves strong performance across the broader benchmark suite.

1 INTRODUCTION

Recent advances in large language models (LLMs) have substantially improved their reasoning capabilities across a broad range of tasks (Guo et al., 2025; Singh et al., 2025; MiniMax et al., 2025), with mathematical reasoning becoming an important setting for evaluating complex multi-step reasoning (Wang et al., 2025a; Raiyan et al., 2026). Despite this progress, LLMs continue to exhibit recurring reasoning errors across problems that share similar underlying structures. Such failures limit the accumulation of reliable problem-solving experience, while even successful reasoning that does not address the underlying failure mode may provide limited reusable guidance for future problems. Critical reasoning constraints can instead serve as compact and reusable guidance, enabling the model to verify intermediate reasoning and steer away from previously observed failure modes. As shown in Figure 1, two nondimensionalization problems with different polynomial degrees exhibit the same structural failure, where the model assumes $a_3 = 1$ and omits the scaling required to normalize the constant term. Enforcing such joint normalization relations $a_3 = M$, $a_2 = L^n M$, and $a_1 = \epsilon L^m M$ prevents this independent normalization and guides the model toward the correct scaling structure.

The effectiveness of a reasoning constraint, however, depends on its applicability to the current problem. A constraint derived from a related problem may capture a transferable reasoning principle, provide only partial guidance, or be irrelevant despite semantic similarity. Consequently, effective constraint reuse requires more than simply attaching retrieved experience to the prompt, where the system must determine whether a retrieved constraint should be examined, reused, or disregarded, and whether a new constraint should be acquired when existing experience is insufficient. This motivates a *constraint-oriented agentic workflow*, in which the constraint itself becomes a central object of orchestration. Rather than treating constraints as fixed auxiliary context, the orchestrator

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

sequentially decides how they should enter the reasoning process, whether to activate a retrieved constraint, construct a new one, invoke a solver under the active constraint, or revise a failed solution while preserving the relevant constraint. The active constraint further determines how reasoning should be verified, allowing the workflow to route problems to natural-language or programmatic operators when appropriate. In this way, the workflow organizes heterogeneous reasoning operations around the acquisition, application, verification, and reuse of constraints, with the utility of these decisions ultimately reflected in the correctness of the final solution.

To this end, we propose **CROW** (Constraint Reuse and Orchestration Workflow), a constraint-guided agentic framework for mathematical reasoning. CROW retrieves related prior experience and allows an orchestrator to determine whether the associated critical reasoning constraint should be inspected, whether reasoning should proceed without constraint guidance, or whether a new constraint should be constructed. When selected, a constraint guides both solution generation and verification through natural-language or program-based reasoning, with revision available when an intermediate execution or reasoning step fails. We formulate these workflow decisions as a sequential policy and optimize the orchestrator with reinforcement learning using downstream solution correctness as the reward signal. This formulation allows the policy to learn when existing constraint knowledge is useful, when direct reasoning is sufficient, and when additional constraint knowledge should be acquired. CROW further retains newly proposed constraints that contribute to successful solutions as reusable experience. The resulting memory makes reasoning principles discovered on earlier training problems available to subsequent problems with related structure, reducing repeated reconstruction of useful constraints. Experience accumulation preserves successful constraint knowledge, while policy optimization determines when such knowledge should influence the current reasoning process. CROW therefore integrates constraint-guided reasoning, adaptive workflow orchestration, and selective experience reuse within a unified mathematical reasoning framework. Our contributions are summarized as follows:

1. We introduce **CROW**, a constraint-guided agentic workflow that treats critical reasoning constraints as reusable knowledge for guiding and verifying mathematical reasoning, while adaptively determining their use for each problem.
2. We formulate constraint utilization and reasoning orchestration as a sequential decision-making problem and optimize the workflow with reinforcement learning, enabling adaptive selection among constraint reuse, direct solving, constraint acquisition, and revision.
3. We introduce an experience accumulation mechanism that retains successful newly proposed constraints for future reuse. Experiments across multiple mathematical reasoning benchmarks demonstrate substantial improvements on HARDMath and competitive performance on other benchmarks, while ablation studies validate the contributions of constraint guidance, workflow optimization, and accumulated experience.

2 Related Work

LLM Reasoning. Recent advances in LLM reasoning have introduced a range of mechanisms for improving complex multi-step problem solving. Natural-language approaches elicit intermediate reasoning through chain-of-thought and structured decomposition (Wei et al., 2022; Huang et al., 2024), while verification-based methods assess intermediate reasoning. to identify erroneous steps (Lightman et al., 2024). A complementary direction externalizes reasoning into executable or symbolic representations, including generated programs (Austin et al., 2021; Chen et al., 2023; Gao et al., 2023; Li et al., 2024), symbolic solvers (Ye et al., 2023; Nguyen et al., 2024; Dutta et al., 2024), and formal reasoning systems (Xin et al., 2024; 2025; Ren et al., 2025). Recent work further explores adaptive selection between natural-language and executable reasoning according to the requirements of individual problems (Chen et al., 2025). Several approaches are designed specifically for mathematical reasoning, including program-assisted mathematical problem solving (Imani et al., 2023; Wang et al., 2024), symbolic mathematical reasoning (Nguyen et al., 2024), and formal theorem proving (Xin et al., 2024; 2025; Ren et al., 2025). These methods improve mathematical reasoning through more reliable computation, structured representations, or formal verification. Nevertheless, mathematical correctness often depends on critical conditions governing intermediate deductions, such as valid transformations, completeness, or problem-specific invariants. When these conditions are not made explicit, locally plausible reasoning can still lead to invalid solutions.

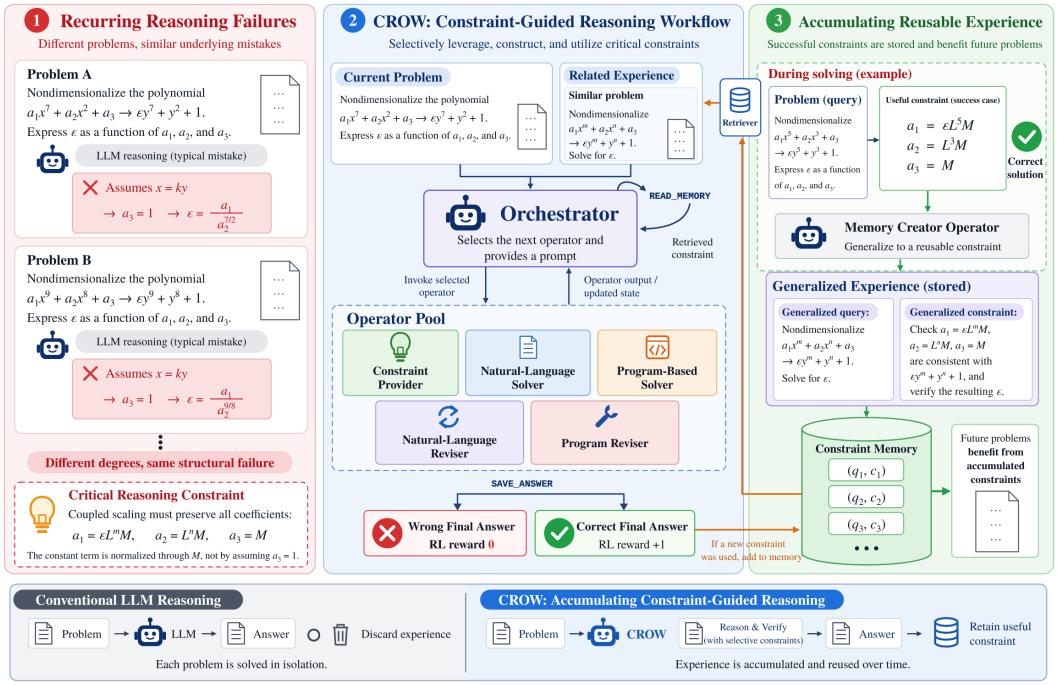

Figure 1: **Overview of CROW.** CROW organizes mathematical reasoning around critical reasoning constraints. An orchestrator iteratively selects operators to retrieve, construct, apply, verify, and revise constraint-guided reasoning, with final-answer correctness providing the RL reward signal. Successful newly constructed constraints are generalized and stored for future problems.

CROW represents such conditions as critical reasoning constraints for guiding and verifying the reasoning process.

Agentic Workflow Optimization.. Agentic workflows decompose complex reasoning into coordinated operations such as planning, tool invocation, execution, verification, and refinement. Existing approaches include static workflows whose structures are optimized during training (Zhang et al., 2025a; Zhao et al., 2026; Wei et al., 2026), as well as dynamic workflows that adapt operator selection and routing according to the evolving reasoning state (Kong et al., 2026; Zhang et al., 2026; Ruan et al., 2026; Su et al., 2025). Recent work further explores evolutionary optimization, where workflow structures, prompts, or agent configurations are iteratively refined according to task-level feedback (Liu et al., 2025; Wang et al., 2025b; Hu et al., 2026). Such decomposition also facilitates error recovery, as execution failures or unsuccessful intermediate outputs can be localized and revised through dedicated verification or debugging components (Huang et al., 2023; Chen et al., 2024; Zhong et al., 2024; Zhang et al., 2025b; Basarkar et al., 2026). These methods primarily optimize the composition and execution of reasoning operators. CROW instead focuses on the orchestration of critical reasoning constraints within the workflow, allowing the policy to determine whether retrieved constraint knowledge should be inspected, applied, bypassed, or replaced by a newly constructed constraint. This extends workflow optimization from selecting and refining reasoning operations to controlling how critical reasoning constraints influence the problem-solving process.

Memory and Experience Evolution.. Memory-augmented agents extend reasoning beyond individual episodes by retaining reusable knowledge from prior interactions, including reflections, experiential summaries, procedural workflows, and modular memories (Shinn et al., 2023; Zhao et al., 2024; Wang et al., 2025c; Han et al., 2026; Yan et al., 2026). More recent work further explores experience evolution, where accumulated knowledge is progressively updated, abstracted, or coupled with policy improvement as additional tasks are solved (Suzgun et al., 2026; Ouyang et al., 2026; Wu et al., 2026). CROW differs in both the representation and utilization of accumulated experience. Rather than retaining complete trajectories, procedural workflows, or general reasoning

strategies, CROW stores critical reasoning constraints that specify conditions governing the validity of mathematical reasoning. These constraints serve as reusable guidance for both reasoning and verification, while their utilization is determined by the learned orchestration policy.

3 METHODOLOGY

3.1 OVERVIEW

CROW treats critical reasoning constraints as reusable knowledge across structurally related problems. As the applicability of a retrieved constraint may vary across problems, its utilization is determined dynamically within the reasoning process. To support such transfer, CROW maintains a constraint memory $\mathcal{M} = \{(q_i, c_i)\}_{i=1}^{N}$, initialized with a small set of seed problem-constraint pairs and progressively expanded during training. Given a new problem q, related experience is retrieved from $\mathcal{M}$ as a candidate source of constraint knowledge rather than being directly imposed on the reasoning process. CROW formulates constraint utilization as a sequential decision process governed by an orchestrator. Conditioned on the current reasoning state, the orchestrator selects the next workflow action and provides the corresponding instruction, allowing constraint retrieval, construction, reasoning, verification, and revision to be coordinated adaptively rather than through a fixed pipeline. The orchestration policy is optimized with reinforcement learning (RL) using final-answer correctness as reward. When a newly constructed constraint contributes to a correct solution, it is generalized and added to memory, enabling successful constraint knowledge to support subsequent problems.

3.2 CONSTRAINT-GUIDED AGENTIC WORKFLOW

CROW treats constraint utilization as part of a broader agentic reasoning process rather than as a fixed preprocessing step. The utility of critical reasoning constraints is problem-dependent: some problems can leverage previously acquired constraints, others require new constraint construction, while simpler cases may be solved without explicit constraint guidance. Moreover, intermediate reasoning may fail even when the underlying approach remains valid, requiring targeted repair rather than restarting the solution process. CROW therefore uses an orchestrator to determine which reasoning operation should be performed next, how constraint knowledge should participate in that operation, and when revision is required to recover from intermediate failures.

Constraint-Oriented Orchestration. The orchestrator repeatedly selects the next workflow action conditioned on the current state of the reasoning process. An action specifies both the operation to perform and the instruction supplied to the corresponding operator. We denote this decision as $a_t = (o_t, p_t)$, where o_t denotes the selected workflow operation or operator and p_t denotes the generated instruction. The allowable actions include accessing retrieved memory, invoking a constraint provider, calling either the natural-language or program-based solver, invoking the corresponding reviser, and submitting the final answer. For solver actions, the orchestrator additionally determines whether the currently available constraint should be supplied as guidance. The invoked operator returns an output that updates the workflow state, which is subsequently provided back to the orchestrator for the next decision. This iterative interaction allows CROW to adapt the reasoning process to intermediate results rather than following a predetermined sequence of operations.

Constraint Memory and Selective Activation. Given a problem q, CROW uses a lightweight retriever $\mathcal{R}$ to identify potentially relevant experience from the constraint memory $\mathcal{M}$. The retriever computes semantic similarity between q and the stored problems $\{q_i\}_{i=1}^{N}$ and returns the most relevant entry, $m^* = (q^*, c^*)$, where q^* is the retrieved problem and c^* is its associated critical reasoning constraint. Retrieval itself does not imply constraint utilization. Only q^* is initially exposed to the orchestrator, allowing it to assess whether the retrieved experience is relevant to the current problem. If the experience is considered applicable, the orchestrator may explicitly access c^* through a memory-read action. This separation between retrieval and activation prevents semantic similarity alone from determining whether a constraint should influence the solution.

Constraint Construction. Retrieved experience may be insufficient when no relevant constraint is available or when the existing constraint does not adequately capture the reasoning requirement

of the current problem. In such cases, the orchestrator can invoke the constraint provider. The orchestrator generates an instruction specifying what aspect of the current reasoning problem should be analyzed, while the constraint provider derives the corresponding critical reasoning constraint. The resulting constraint is returned to the workflow state and can subsequently be supplied to solving or revision operators. Thus, constraint construction remains a specialized reasoning operation, while the orchestrator determines when it should be invoked and how it should be conditioned.

Constraint-Guided Reasoning. Accessing a constraint does not require it to govern the subsequent solution. The suitability of constraint guidance and the preferred reasoning modality may both vary across problems. CROW therefore allows the orchestrator to invoke either the natural-language solver or the program-based solver, with or without providing the active constraint. These choices are learned by the orchestrator through reinforcement learning rather than specified by fixed routing heuristics. When constraint guidance is used, the solver is required to derive the solution consistently with the active constraint and to verify the resulting candidate against its stated requirements. Natural-language reasoning supports constraints that are naturally expressed and checked through mathematical derivation, while programmatic reasoning enables symbolic computation and executable verification when appropriate. The two modalities are therefore complementary mechanisms for operationalizing the same constraint-oriented reasoning process rather than separate fixed branches of the workflow. Detailed examples are provided in Appendices G.1 and G.2.

Constraint-Preserving Revision. Intermediate solver outputs may remain recoverable despite malformed answer formatting, incomplete reasoning, execution errors, or failed verification. CROW therefore includes modality-specific revision operators. The natural-language reviser repairs unsuccessful natural-language responses, while the program reviser diagnoses and repairs executable programs using the observed execution or verification feedback. When the preceding attempt uses an active constraint, the same constraint is made available during revision so that the repair remains consistent with the intended reasoning requirement. The revised output is returned to orchestrator rather than being automatically accepted, allowing orchestrator to determine whether another operation is required or whether final answer should be submitted. A concrete example of verifier-guided program repair is provided in Appendix H.1.

3.3 Reinforcement Learning for Workflow Orchestration

Given the workflow state s_t at step t, the objective of the orchestrator is to select an action a_t that contributes to a correct final solution. We parameterize the orchestrator as a policy $\pi_\theta(a_t \mid s_t)$ and optimize it with RL. A trajectory $\tau = (s_0, a_0, \ldots, s_T, a_T)$ terminates with a predicted answer $\hat{y}$. We use a sparse terminal reward, $R(\tau) = 1$ if $\hat{y}$ is correct and $R(\tau) = 0$ otherwise. This allows the policy to evaluate intermediate orchestration decisions through their downstream effect on final-answer correctness. The RL objective therefore does not require explicit supervision over the desired action at each step. Instead, the orchestrator learns which action to take under the current workflow state from the success or failure of complete reasoning trajectories.

3.4 Experience Accumulation

Policy optimization determines how the orchestrator navigates the workflow, whereas successful constraint discovery provides additional reusable knowledge for subsequent problems. Without retaining such discoveries, useful reasoning principles identified through PROPOSE would need to be repeatedly reconstructed. CROW therefore stores newly proposed constraints that contribute to correct solutions. Specifically, if a proposed constraint c_q is used in a successful trajectory for problem q, the memory is updated as $\mathcal{M} \leftarrow \mathcal{M} \cup \{(q, c_q)\}$. The resulting memory entry becomes available for retrieval in later problems with related structure, allowing verified constraint knowledge to be transferred across training instances. Experience accumulation and policy optimization serve complementary roles. The former preserves successful constraint knowledge, while the latter learns whether such knowledge should be reused for a given problem. This design reduces repeated constraint construction while maintaining selective use of the accumulated memory.

Table 1: Accuracy (%) across mathematical reasoning benchmarks. Best results are shown in bold.

Category	Method	HARDMath	HARDMath2	Precalculus	Calculus	Algebra	Overall
Base	Text	2.60	4.00	79.82	71.74	58.45	48.64
	Code	1.62	4.50	63.39	42.39	57.75	37.95
Test-Time	Plan and Solve	2.60	4.50	78.75	64.67	55.63	47.06
	Self-Refine	2.92	5.00	82.32	70.65	56.34	49.50
	Multi-Agent Debate	3.25	5.50	89.11	75.00	64.79	53.80
Search-Based Workflow	AFlow	4.22	5.50	89.11	77.72	63.38	54.23
	ADAS	2.60	5.50	83.21	76.09	54.93	50.43
	MASS	3.57	5.50	90.18	77.72	61.97	54.38
	EvoMAS	1.95	4.50	73.39	57.07	64.08	44.62
Learning-Based Workflow	Prompt-R1	2.60	4.50	87.14	80.43	60.56	53.01
	AgentFlow	2.60	2.00	83.39	77.72	57.04	50.43
	The Conductor	3.90	**10.00**	88.93	**82.61**	63.38	55.38
	CROW	**30.84**	5.50	**91.43**	78.80	**65.49**	**61.41**

4 EXPERIMENTS

Benchmarks. We evaluate on challenging mathematical reasoning problems drawn from three sources: HARDMath (Fan et al., 2025), HARDMath2 (Roggeveen et al., 2025), and DeepMath-103K (He et al., 2026). HARDMath and HARDMath2 are designed to evaluate difficult applied mathematical reasoning. From DeepMath-103K, we specifically select three challenging subject categories: **Precalculus**, **Calculus**, and **Algebra**, yielding five evaluation sets in total. We construct 1,748 training examples and use a held-out test set of 1,394 problems for evaluation.

Experimental Setup. We use Qwen3-Coder-30B-A3B-Instruct .[1] as the task-solving backbone for all experiments For learning-based workflow methods, we use Qwen3-4B (Yang et al., 2025) [2] as the orchestrator model. For our memory retrieval, we use the frozen Qwen3-Embedding-0.6B [3] model. For learning-based workflow methods, the orchestrator is trained on the 1,748 training examples, while the Qwen3-Coder-30B backbone remains fixed. Search-based methods use the same training set for workflow optimization without updating model parameters. The test set is strictly held out from both workflow optimization and policy training and is used only for final evaluation. We report accuracy as the primary evaluation metric. For final evaluation, we extract the final answer from each model response and use GPT-4o as the judge to ensure consistent scoring across all methods. During training and workflow optimization, we instead use Qwen3-Coder-30B as the judge to reduce evaluation cost. Detailed prompts, decoding settings, optimization procedures, and training hyperparameters are provided in Appendix A.

Baselines. We compare against four categories of baselines. **Base** methods directly solve each problem using either text-based or code-based reasoning. **Test-Time** methods improve reasoning at inference time without optimizing a persistent workflow, including Self-Refine (Madaan et al., 2023), Self-Consistency (Wang et al., 2023), and Multi-Agent Debate (Du et al., 2024). **Search-Based Workflow** methods optimize workflow structures or prompts through search without updating model parameters, including AFlow, ADAS (Hu et al., 2025), MASS (Zhou et al., 2026), and EvoMAS (Hu et al., 2026). **Learning-Based Workflow** methods learn an orchestration policy through parameter optimization, including Prompt-R1 (Liu et al., 2026), AgentFlow (Li et al., 2026), and The Conductor (Nielsen et al., 2026). Detailed implementation and adaptation settings for all baselines are provided in Appendix A.

5 RESULTS

5.1 MAIN RESULTS

Overall performance. CROW achieves the best aggregate accuracy, reaching 61.41% and outperforming the strongest baseline, The Conductor, by 6.03%. The largest improvement occurs on

[1] Due to storage constraints, we use the AWQ-quantized checkpoint available at `https://huggingface.co/stelterlab/Qwen3-Coder-30B-A3B-Instruct-AWQ`.

[2] `https://huggingface.co/Qwen/Qwen3-4B`

[3] `https://huggingface.co/Qwen/Qwen3-Embedding-0.6B`

HARDMath, where CROW reaches 30.84%, compared with 4.22% or lower for all baselines. Beyond HARDMath, CROW maintains competitive performance across the remaining benchmarks, with modest improvements on Precalculus and Algebra. These results suggest that CROW primarily extends the range of difficult symbolic problems that can be solved while preserving strong performance on more conventional mathematical reasoning tasks.

Limitations of Test-Time and Search-Based Methods. Test-time methods improve reasoning through planning, revision, or solution diversity, but remain limited when the same reasoning error persists across candidates. As shown in Appendix D, Plan-and-Solve may reproduce errors already present in the base model, Self-Refine can recognize a defect yet introduce a new one during unverified revision, and Multi-Agent Debate offers limited correction when agents converge on the same flawed reasoning. Search-based workflows expand candidate exploration through proposal, reflection, refinement, and aggregation, but repeated selection cannot recover a missing reasoning relation if it is absent from the explored trajectories. Their gains therefore primarily improve exploration and local correction rather than resolving shared reasoning failures.

Limitations of Learning-Based Workflows. Learning-based workflows improve prompting, decomposition, and routing, but still depend on the reasoning strategies available to the underlying solver. Prompt-R1 may reinforce an incorrect premise through its learned interaction prompt, AgentFlow may detect an incomplete response without introducing the missing reasoning relation, and The Conductor can propagate a shared error across downstream workers when their inputs inherit the same flawed premise. Representative trajectories are provided in Appendix D. Thus, learned orchestration improves the elicitation and coordination of existing reasoning capabilities, but does not reliably recover reasoning knowledge absent from the generated trajectory.

Effect of Constraint Knowledge from CROW. CROW addresses these limitations by making critical reasoning constraints explicit, reusable components of the reasoning process. Rather than relying on additional search or repeated prompting to rediscover a missing reasoning relation, CROW can retrieve a constraint acquired from a structurally related problem and selectively incorporate it into the current trajectory. Once activated, the constraint does more than provide additional context: it explicitly restricts the reasoning process and steers the solver away from previously observed erroneous reasoning patterns. The same constraint also provides a common specification for solution generation, verification, and revision, while the orchestrator learns whether it should be used and which reasoning operator is appropriate. This allows CROW to correct recurring failures that may otherwise persist across independently generated candidates or coordinated agents. The effect is most pronounced on HARDMath, where the substantial improvement is associated with transferring reusable constraint knowledge across related problems rather than increasing search depth or the number of agent interactions.

5.2 Error Analysis

CROW exhibits three general failure modes, illustrated by complete trajectories in Appendix I. First, despite being trained to assess constraint applicability, the orchestrator may still accept a constraint from a superficially similar problem whose assumptions do not transfer. Second, when no applicable constraint exists, the policy may reject the retrieved candidate but fail to construct a useful replacement, causing the workflow to fall back to direct reasoning. Third, newly constructed constraints may themselves be mathematically incorrect and consequently misguide the subsequent solution. These failure modes help explain why CROW does not achieve the best performance on every benchmark. In HARDMath2, the limited amount of training data provides relatively few opportunities to acquire reusable constraints during training. In Algebra, many acquired constraints exhibit limited transferability across queries, reducing the effectiveness of constraint reuse.

5.3 Inference Efficiency

Table 4 reports the estimated Qwen3-Coder-30B inference cost over the same 50-query sample. Appendix C describes the estimation procion proc. Methods such as Multi-Agent Debate, AFlow, MASS, ADAS, and EvoMAS rely on fixed or repeatedly invoked multi-stage workflows involving multiple candidate generations, refinement, selection, or verification steps, leading to substantially

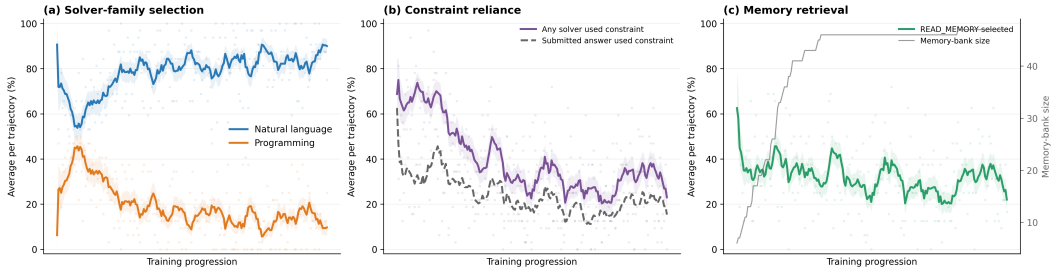

Figure 2: Evolution of CROW's orchestration policy during training. The plots report the average trajectory-level utilization of natural-language and programming solvers, reasoning constraints, and memory retrieval. CROW progressively converges toward direct natural-language reasoning while using programming, constraints, and memory more selectively.

Table 2: Ablation study of the proposed method across five mathematical reasoning benchmarks.

Constraint	Memory	RL	HARDMath	HARDMath2	Precalculus	Calculus	Algebra	Overall
×	×	×	5.52	5.00	85.89	71.20	59.15	51.87
×	×	✓	6.49	3.50	88.75	**79.35**	52.82	53.44
✓	×	×	1.30	0.50	47.68	32.07	55.63	29.41
✓	×	✓	1.95	4.50	68.75	49.46	57.75	41.10
✓	✓	×	24.03	2.50	63.93	46.74	59.15	43.54
✓	✓	✓	**30.84**	**5.50**	**92.43**	78.80	**65.49**	**61.41**

higher inference cost. Notably, EvoMAS has a particularly large estimated token cost of 184.2K tokens per query. CROW instead uses a learned orchestrator to conditionally invoke memory access, constraint construction, programming, and revision according to the current reasoning state. By avoiding a fixed sequence of auxiliary operations for every problem, CROW achieves a favorable accuracy-cost trade-off among agentic workflows.

5.4 Training Progression

We analyze how the orchestration policy changes throughout training in Figure 2. Each operation is counted at most once per trajectory, while trajectories that use both natural-language and programming solvers contribute to both categories. Early training exhibits substantial exploration: programming becomes more frequent as natural-language reasoning temporarily declines. This trend later reverses, with the policy converging toward natural-language reasoning for most trajectories and invoking the programmer only selectively (Figure 2a). Constraint and memory usage follow a similar transition toward selectivity. Both the use of constraints during solving and their inclusion in submitted answers decline over training (Figure 2b), suggesting that the policy learns to avoid unnecessary guidance. Memory retrieval also decreases and stabilizes despite the continued growth of the memory bank (Figure 2c). Collectively, these trends indicate that training does not encourage indiscriminate use of every available component. Instead, CROW learns to rely primarily on direct reasoning while retaining programming, constraints, and memory as targeted support mechanisms.

6 Ablation Study

Table 2 examines the contribution of constraint generation, memory, and RL. The results indicate complementary roles: revision improves workflow robustness, memory supplies reusable reasoning knowledge, and RL helps orchestrator to learn when different reasoning resources should be invoked. Appendix E details the prompt, action-space, and component changes used to construct each variant.

Workflow without constraints, memory, and RL. The component-free workflow retains the benefit of modality-specific revision. Malformed natural-language responses and failed programs can be repaired rather than immediately treated as unsuccessful, resulting in a modest robustness gain over direct inference without introducing additional reasoning knowledge.

RL without constraints and memory. RL enables adaptive routing between natural-language and programmatic reasoning, but in the absence of additional reasoning resources the learned policy tends to favor natural-language reasoning to maximize training reward. Programmatic reasoning is retained only for a limited subset of harder problems. This suggests that workflow optimization can improve modality selection, but may also collapse toward the dominant reasoning mode when the available operators provide only limited complementary benefit.

Constraints without memory and RL. Introducing constraints without memory or learned orchestration substantially degrades performance. In this setting, the workflow frequently constructs and applies constraints even when they are unnecessary or unreliable, which can introduce incorrect assumptions, unsuitable verification criteria, and additional execution failures. These results indaticate that constraint guidance is effective only when supported by reliable constraint knowledge and selective orchestration.

RL with constraints but without memory. Adding RL partially recovers the degradation caused by constraint-only reasoning, but the learned policy still relies heavily on constraint-guided programming. The improvement is driven primarily by more effective revision and recovery from failed executions rather than by avoiding unreliable constraints altogether. This suggests that RL can improve the robustness of constraint-guided reasoning, but cannot substitute for reliable reusable constraint knowledge.

Memory without RL. Adding memory substantially improves HARDMath by providing reusable, previously validated constraints that transfer across related problems. However, without RL the workflow still overuses constraint-guided reasoning on easier benchmarks, where direct solving is often more appropriate. Memory therefore improves the availability of reliable constraint knowledge, but does not by itself determine when that knowledge should be used.

Complete system. Combining constraints, memory, and RL yields the strongest overall performance. Memory provides reusable reasoning constraints that are particularly effective on HARDMath, while RL enables selective orchestration so that constraint-guided reasoning is invoked when beneficial and direct solving is preserved when additional guidance is unnecessary. The complete system therefore combines reliable constraint knowledge with learned control over when and how that knowledge should be used. Overall, the ablation shows that revision improves workflow robustness, memory supplies reusable knowledge for difficult problems, and RL provides the selective orchestration needed to determine when and how that knowledge should be used.

7 Conclusion

We introduced CROW, a learned orchestration framework that selectively coordinates memory retrieval, constraint construction, solver modality, and revision around a frozen foundation model. Our results show that test-time refinement and static workflow search primarily help when the solver already possesses the necessary knowledge, whereas difficult symbolic reasoning benefits from transferring validated strategies across structurally related problems. The ablations further demonstrate that constraints can be harmful in isolation: memory provides reliable reasoning experience, while reinforcement learning enables the orchestrator to decide when to retrieve it and when direct reasoning is more appropriate. Overall, CROW shows that robust reasoning depends not merely on additional inference, but on learning to selectively reuse and verify accumulated problem-solving knowledge. Future work can extend CROW to broader reasoning domains, improve how successful constraints are generalized before being stored to reduce over-specificity to individual queries, and strengthen on-the-fly constraint proposal when retrieved memory is insufficient. Future work can extend CROW to broader reasoning domains, improve how successful constraints are generalized before being stored to reduce over-specificity to individual queries, and strengthen on-the-fly constraint proposal when retrieved memory is insufficient or only partially applicable.

AI use statement

In this work, we used generative AI tools to assist with method implementation through coding and debugging, clean and reformat experimental data and outputs, evaluate model outputs, analyze and

cluster recurring error cases in both baseline and proposed methods, and support the interpretation of experimental results. In particular, generative AI was used to help identify common mathematical reasoning failures exhibited by current models. These candidate failure patterns were subsequently inspected and verified by the authors against the original model outputs before being used to characterize model weaknesses and inform the motivation and refinement of the proposed methodology. Generative AI was not used to derive mathematical proofs or establish formal mathematical results for the proposed method. We also did not use generative AI to generate synthetic datasets or perform translation, datasets and benchmarks were selected by the authors from existing public sources.

Generative AI was additionally used to provide limited feedback on experimental configurations, create and edit software code, draft and revise parts of the manuscript, improve readability and organization, summarize and discuss relevant literature, format LaTeX content, assist with formatting bibliographic entries and citations, and create or modify scientific figures and plots. The papers cited in this work were identified and selected by the authors, who read the original sources, and generative AI was used only to help express the corresponding references in the required format. Experimental settings were primarily chosen based on prior work and our own experimental design to ensure fair comparison, AI suggestions were considered only when an existing configuration was found to be unsuitable. AI-assisted figures and plots were constructed from the actual experimental results and the authors' description of the proposed method. The core research idea, methodological contribution, and claimed novelty were developed by the authors.

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Aditya Basarkar, Benyamin Tabarsi, Tiffany Barnes, and Dongkuan Xu. Enhancing mathematical problem solving in llms through execution-driven reasoning augmentation. *arXiv preprint arXiv:2602.03950*, 2026.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research (TMLR)*, 2023.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

Yongchao Chen, Yilun Hao, Yueying Liu, Yang Zhang, and Chuchu Fan. : Symbolic-augmented language models via code/text guidance. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2025.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2024.

Subhabrata Dutta, Joykirat Singh, Ishan Pandey, Sunny Manchanda, Soumen Chakrabarti, and Tanmoy Chakraborty. Frugal lms trained to invoke symbolic solvers achieve parameter-efficient arithmetic reasoning. In *Proceedings of the Association for the Advancement of Artificial Intelligence (AAAI)*, 2024.

Jingxuan Fan, Sarah Martinson, Erik Y. Wang, Kaylie Hausknecht, Jonah Brenner, Danxian Liu, Nianli Peng, Corey Wang, and Michael P. Brenner. HARDMath: A benchmark dataset for challenging problems in applied mathematics. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2023.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Nature*, 2025.

Dongge Han, Camille Couturier, Daniel Madrigal Diaz, Xuchao Zhang, Victor Rühle, and Saravan Rajmohan. Legomem: Modular procedural memory for multi-agent llm systems for workflow automation. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2026.

Zhiwei He, Tian Liang, Jiahao Xu, Qiuzhi Liu, Xingyu Chen, Yue Wang, Linfeng Song, Dian Yu, Zhenwen Liang, Wenxuan Wang, Zhuosheng Zhang, Rui Wang, Zhaopeng Tu, Haitao Mi, and Dong Yu. DeepMath-103K: A large-scale, challenging, decontaminated, and verifiable mathematical dataset for advancing reasoning. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2026.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025.

Yuntong Hu, Yuting Zhang, Matthew Trager, Yi Zhang, Shuo Yang, Wei Xia, and Stefano Soatto. EvoMAS: Evolutionary generation of multi-agent systems. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2026.

Dong Huang, Qingwen Bu, Jie M. Zhang, Michael Luck, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.

Xu Huang, Weiwen Liu, Xiaolong Chen, Xingmei Wang, Hao Wang, Defu Lian, Yasheng Wang, Ruiming Tang, and Enhong Chen. Understanding the planning of llm agents: A survey. *arXiv preprint arXiv:2402.02716*, 2024.

Shima Imani, Liang Du, and Harsh Shrivastava. Mathprompter: Mathematical reasoning using large language models. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL) Volume 5: Industry Track*, 2023.

Mingze Kong, Zikun Qu, Zhongquan Zhou, Pengyu Liang, Xiang Li, Zhiwei Shang, Zhi Hong, Kaiyu Huang, Zhiyong Wang, and Zhongxiang Dai. Workflow-r1: Group sub-sequence policy optimization for multi-turn workflow construction. *arXiv preprint arXiv:2602.01202*, 2026.

Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. Chain of code: Reasoning with a language model-augmented code emulator. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2024.

Zhuofeng Li, Haoxiang Zhang, Seungju Han, Sheng Liu, Jianwen Xie, Yu Zhang, Yejin Choi, James Zou, and Pan Lu. In-the-flow agentic system optimization for effective planning and tool use. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2026.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

Siwei Liu, Jinyuan Fang, Han Zhou, Yingxu Wang, and Zaiqiao Meng. Sew: Self-evolving agentic workflows for automated code generation. *arXiv preprint arXiv:2505.18646*, 2025.

Wenjin Liu, Haoran Luo, Xueyuan Lin, Haoming Liu, Tiesunlong Shen, Jiapu Wang, Rui Mao, and Erik Cambria. Prompt-R1: Collaborative automatic prompting framework via end-to-end reinforcement learning. In *Findings of the Association for Computational Linguistics: (ACL)*, 2026.

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. Self-refine: Iterative refinement with self-feedback. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

MiniMax, :, Aili Chen, Aonian Li, Bangwei Gong, Binyang Jiang, Bo Fei, Bo Yang, Boji Shan, Changqing Yu, Chao Wang, Cheng Zhu, Chengjun Xiao, Chengyu Du, Chi Zhang, Chu Qiao, Chunhao Zhang, Chunhui Du, Congchao Guo, Da Chen, Deming Ding, Dianjun Sun, Dong Li, Enwei Jiao, Haigang Zhou, Haimo Zhang, Han Ding, Haohai Sun, Haoyu Feng, Huaiguang Cai, Haichao Zhu, Jian Sun, Jiaqi Zhuang, Jiaren Cai, Jiayuan Song, Jin Zhu, Jingyang Li, Jinhao Tian, Jinli Liu, Junhao Xu, Junjie Yan, Junteng Liu, Junxian He, Kaiyi Feng, Ke Yang, Kecheng Xiao, Le Han, Leyang Wang, Lianfei Yu, Liheng Feng, Lin Li, Lin Zheng, Linge Du, Lingyu Yang, Lunbin Zeng, Minghui Yu, Mingliang Tao, Mingyuan Chi, Mozhi Zhang, Mujie Lin, Nan Hu, Nongyu Di, Peng Gao, Pengfei Li, Pengyu Zhao, Qibing Ren, Qidi Xu, Qile Li, Qin Wang, Rong Tian, Ruitao Leng, Shaoxiang Chen, Shaoyu Chen, Shengmin Shi, Shitong Weng, Shuchang Guan, Shuqi Yu, Sichen Li, Songquan Zhu, Tengfei Li, Tianchi Cai, Tianrun Liang, Weiyu Cheng, Weize Kong, Wenkai Li, Xiancai Chen, Xiangjun Song, Xiao Luo, Xiao Su, Xiaobo Li, Xiaodong Han, Xinzhu Hou, Xuan Lu, Xun Zou, Xuyang Shen, Yan Gong, Yan Ma, Yang Wang, Yiqi Shi, Yiran Zhong, Yonghong Duan, Yongxiang Fu, Yongyi Hu, Yu Gao, Yuanxiang Fan, Yufeng Yang, Yuhao Li, Yulin Hu, Yunan Huang, Yunji Li, Yunzhi Xu, Yuxin Mao, Yuxuan Shi, Yuze Wenren, Zehan Li, Zelin Li, Zhanxu Tian, Zhengmao Zhu, Zhenhua Fan, Zhenzhen Wu, Zhichao Xu, Zhihang Yu, Zhiheng Lyu, Zhuo Jiang, Zibo Gao, Zijia Wu, Zijian Song, and Zijun Sun. Minimax-m1: Scaling test-time compute efficiently with lightning attention. *arXiv preprint arXiv:2506.13585*, 2025.

Minh Phu Nguyen, Minh Phuong Pham, Man Ngo, and Kha Tuan Minh. Math2sym: A system for solving elementary problems via large language models and symbolic solvers. 2024.

Stefan Nielsen, Edoardo Cetin, Peter Schwendeman, Qi Sun, Jinglue Xu, and Yujin Tang. Learning to orchestrate agents in natural language with the conductor. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2026.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2026.

Syed Rifat Raiyan, Mohsinul Kabir, Hasan Mahmud, and Md Kamrul Hasan. Artificial intelligence for mathematical reasoning: An integrated survey of language models, neuro-symbolic systems, and verified discovery. *arXiv preprint arXiv:2606.08728*, 2026.

Z.Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z.F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*, 2025.

James V. Roggeveen, Erik Y. Wang, David Ettel, Will Flintoft, Peter Donets, Lucy S. Nathwani, Nickholas Gutierrez, Anton Marius Graf, Siddharth Dandavate, Arjun Nageswaran, Raglan Ward, Ava Williamson, Anne Mykland, Kacper K. Migacz, Yijun Wang, Egemen Bostan, Duy Thuc Nguyen, Zhe He, Marc L. Descoteaux, Felix Yeung, Shida Liu, Jorge García Ponce, Luke Zhu, Yuyang Chen, Ekaterina S. Ivshina, Miguel Fernandez, Minjae Kim, Kennan Gumbs, Matthew Scott Tan, Russell Yang, Mai Hoang, David Brown, Isabella A. Silveira, Lavon Sykes, Ahmed Roman, William Fredenberg, Yiming Chen, Lucas Martin, Yixing Tang, Kelly Werker Smith, Hongyu Liao, Logan G. Wilson, Alexander Dazhen Cai, Andrea Elizabeth Biju, and Michael P. Brenner. HARDMath2: A benchmark for applied mathematics built by students as part of a graduate class. In *Proceedings of the Neural Information Processing Systems (NeurIPS) Track on Datasets and Benchmarks*, 2025.

Jianhao Ruan, Zhihao Xu, Yiran Peng, Fashen Ren, Zhaoyang Yu, Xinbing Liang, Jinyu Xiang, Yongru Chen, Bang Liu, Chenglin Wu, Yuyu Luo, and Jiayi Zhang. Aorchestra: Automating sub-agent creation for agentic orchestration. *arXiv preprint arXiv:2602.03786*, 2026.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Proceedings of the Neural Information Processing Systems (NeurIPS)*, 2023.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, Akshay Nathan, Alan Luo, Alec Helyar, Aleksander Madry, Aleksandr Efremov, Aleksandra Spyra, Alex Baker-Whitcomb, Alex Beutel, Alex Karpenko, Alex Makelov, Alex Neitz, Alex Wei, Alexandra Barr, Alexandre Kirchmeyer, Alexey Ivanov, Alexi Christakis, Alistair Gillespie, Allison Tam, Ally Bennett, Alvin Wan, Alyssa Huang, Amy McDonald Sandjideh, Amy Yang, Ananya Kumar, Andre Saraiva, Andrea Vallone, Andrei Gheorghe, Andres Garcia Garcia, Andrew Braunstein, Andrew Liu, Andrew Schmidt, Andrey Mereskin, Andrey Mishchenko, Andy Applebaum, Andy Rogerson, Ann Rajan, Annie Wei, Anoop Kotha, Anubha Srivastava, Anushree Agrawal, Arun Vijayvergiya, Ashley Tyra, Ashvin Nair, Avi Nayak, Ben Eggers, Bessie Ji, Beth Hoover, Bill Chen, Blair Chen, Boaz Barak, Borys Minaiev, Botao Hao, Bowen Baker, Brad Lightcap, Brandon McKinzie, Brandon Wang, Brendan Quinn, Brian Fioca, Brian Hsu, Brian Yang, Brian Yu, Brian Zhang, Brittany Brenner, Callie Riggins Zetino, Cameron Raymond, Camillo Lugaresi, Carolina Paz, Cary Hudson, Cedric Whitney, Chak Li, Charles Chen, Charlotte Cole, Chelsea Voss, Chen Ding, Chen Shen, Chengdu Huang, Chris Colby, Chris Hallacy, Chris Koch, Chris Lu, Christina Kaplan, Christina Kim, CJ Minott-Henriques, Cliff Frey, Cody Yu, Coley Czarnecki, Colin Reid, Colin Wei, Cory Decareaux, Cristina Scheau, Cyril Zhang, Cyrus Forbes, Da Tang, Dakota Goldberg, Dan Roberts, Dana Palmie, Daniel Kappler, Daniel Levine, Daniel Wright, Dave Leo, David Lin, David Robinson, Declan Grabb, Derek Chen, Derek Lim, Derek Salama, Dibya Bhattacharjee, Dimitris Tsipras, Dinghua Li, Dingli Yu, DJ Strouse, Drew Williams, Dylan Hunn, Ed Bayes, Edwin Arbus, Ekin Akyurek, Elaine Ya Le, Elana Widmann, Eli Yani, Elizabeth Proehl, Enis Sert, Enoch Cheung, Eri Schwartz, Eric Han, Eric Jiang, Eric Mitchell, Eric Sigler, Eric Wallace, Erik Ritter, Erin Kavanaugh, Evan Mays, Evgenii Nikishin, Fangyuan Li, Felipe Petroski Such, Filipe de Avila Belbute Peres, Filippo Raso, Florent Bekerman, Foivos Tsimpourlas, Fotis Chantzis, Francis Song, Francis Zhang, Gaby Raila, Garrett McGrath, Gary Briggs, Gary Yang, Giambattista Parascandolo, Gildas Chabot, Grace Kim, Grace Zhao, Gregory Valiant, Guillaume Leclerc, Hadi Salman, Hanson Wang, Hao Sheng, Haoming Jiang, Haoyu Wang, Haozhun Jin, Harshit Sikchi, Heather Schmidt, Henry Aspegren, Honglin Chen, Huida Qiu, Hunter Lightman, Ian Covert, Ian Kivlichan, Ian Silber, Ian Sohl, Ibrahim Hammoud, Ignasi Clavera, Ikai Lan, Ilge Akkaya, Ilya Kostrikov, Irina Kofman, Isak Etinger, Ishaan Singal, Jackie Hehir, Jacob Huh, Jacqueline Pan, Jake Wilczynski, Jakub Pachocki, James Lee, James Quinn, Jamie Kiros, Janvi Kalra, Jasmyn Samaroo, Jason Wang, Jason Wolfe, Jay Chen, Jay Wang, Jean Harb, Jeffrey Han, Jeffrey Wang, Jennifer Zhao, Jeremy Chen, Jerene Yang, Jerry Tworek, Jesse Chand, Jessica Landon, Jessica Liang, Ji Lin, Jiancheng Liu, Jianfeng Wang, Jie Tang, Jihan Yin, Joanne Jang, Joel Morris,

Joey Flynn, Johannes Ferstad, Johannes Heidecke, John Fishbein, John Hallman, Jonah Grant, Jonathan Chien, Jonathan Gordon, Jongsoo Park, Jordan Liss, Jos Kraaijeveld, Joseph Guay, Joseph Mo, Josh Lawson, Josh McGrath, Joshua Vendrow, Joy Jiao, Julian Lee, Julie Steele, Julie Wang, Junhua Mao, Kai Chen, Kai Hayashi, Kai Xiao, Kamyar Salahi, Kan Wu, Karan Sekhri, Karan Sharma, Karan Singhal, Karen Li, Kenny Nguyen, Keren Gu-Lemberg, Kevin King, Kevin Liu, Kevin Stone, Kevin Yu, Kristen Ying, Kristian Georgiev, Kristie Lim, Kushal Tirumala, Kyle Miller, Lama Ahmad, Larry Lv, Laura Clare, Laurance Fauconnet, Lauren Itow, Lauren Yang, Laurentia Romaniuk, Leah Anise, Lee Byron, Leher Pathak, Leon Maksin, Leyan Lo, Leyton Ho, Li Jing, Liang Wu, Liang Xiong, Lien Mamitsuka, Lin Yang, Lindsay McCallum, Lindsey Held, Liz Bourgeois, Logan Engstrom, Lorenz Kuhn, Louis Feuvrier, Lu Zhang, Lucas Switzer, Lukas Kondraciuk, Lukasz Kaiser, Manas Joglekar, Mandeep Singh, Mandip Shah, Manuka Stratta, Marcus Williams, Mark Chen, Mark Sun, Marselus Cayton, Martin Li, Marvin Zhang, Marwan Aljubeh, Matt Nichols, Matthew Haines, Max Schwarzer, Mayank Gupta, Meghan Shah, Melody Y. Guan, Melody Huang, Meng Dong, Mengqing Wang, Mia Glaese, Micah Carroll, Michael Lampe, Michael Malek, Michael Sharman, Michael Zhang, Michele Wang, Michelle Pokrass, Mihai Florian, Mikhail Pavlov, Miles Wang, Ming Chen, Mingxuan Wang, Minnia Feng, Mo Bavarian, Molly Lin, Moose Abdool, Mostafa Rohaninejad, Nacho Soto, Natalie Staudacher, Natan LaFontaine, Nathan Marwell, Nelson Liu, Nick Preston, Nick Turley, Nicklas Ansman, Nicole Blades, Nikil Pancha, Nikita Mikhaylin, Niko Felix, Nikunj Handa, Nishant Rai, Nitish Keskar, Noam Brown, Ofir Nachum, Oleg Boiko, Oleg Murk, Olivia Watkins, Oona Gleeson, Pamela Mishkin, Patryk Lesiewicz, Paul Baltescu, Pavel Belov, Peter Zhokhov, Philip Pronin, Phillip Guo, Phoebe Thacker, Qi Liu, Qiming Yuan, Qinghua Liu, Rachel Dias, Rachel Puckett, Rahul Arora, Ravi Teja Mullapudi, Raz Gaon, Reah Miyara, Rennie Song, Rishabh Aggarwal, RJ Marsan, Robel Yemiru, Robert Xiong, Rohan Kshirsagar, Rohan Nuttall, Roman Tsiupa, Ronen Eldan, Rose Wang, Roshan James, Roy Ziv, Rui Shu, Ruslan Nigmatullin, Saachi Jain, Saam Talaie, Sam Altman, Sam Arnesen, Sam Toizer, Sam Toyer, Samuel Miserendino, Sandhini Agarwal, Sarah Yoo, Savannah Heon, Scott Ethersmith, Sean Grove, Sean Taylor, Sebastien Bubeck, Sever Banesiu, Shaokyi Amdo, Shengjia Zhao, Sherwin Wu, Shibani Santurkar, Shiyu Zhao, Shraman Ray Chaudhuri, Shreyas Krishnaswamy, Shuaiqi, Xia, Shuyang Cheng, Shyamal Anadkat, Simón Posada Fishman, Simon Tobin, Siyuan Fu, Somay Jain, Song Mei, Sonya Egoian, Spencer Kim, Spug Golden, SQ Mah, Steph Lin, Stephen Imm, Steve Sharpe, Steve Yadlowsky, Sulman Choudhry, Sungwon Eum, Suvansh Sanjeev, Tabarak Khan, Tal Stramer, Tao Wang, Tao Xin, Tarun Gogineni, Taya Christianson, Ted Sanders, Tejal Patwardhan, Thomas Degry, Thomas Shadwell, Tianfu Fu, Tianshi Gao, Timur Garipov, Tina Sriskandarajah, Toki Sherbakov, Tomek Korbak, Tomer Kaftan, Tomo Hiratsuka, Tongzhou Wang, Tony Song, Tony Zhao, Troy Peterson, Val Kharitonov, Victoria Chernova, Vineet Kosaraju, Vishal Kuo, Vitchyr Pong, Vivek Verma, Vlad Petrov, Wanning Jiang, Weixing Zhang, Wenda Zhou, Wenlei Xie, Wenting Zhan, Wes McCabe, Will DePue, Will Ellsworth, Wulfie Bain, Wyatt Thompson, Xiangning Chen, Xiangyu Qi, Xin Xiang, Xinwei Shi, Yann Dubois, Yaodong Yu, Yara Khakbaz, Yifan Wu, Yilei Qian, Yin Tat Lee, Yinbo Chen, Yizhen Zhang, Yizhong Xiong, Yonglong Tian, Young Cha, Yu Bai, Yu Yang, Yuan Yuan, Yuanzhi Li, Yufeng Zhang, Yuguang Yang, Yujia Jin, Yun Jiang, Yunyun Wang, Yushi Wang, Yutian Liu, Zach Stubenvoll, Zehao Dou, Zheng Wu, and Zhigang Wang. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

Hongjin Su, Shizhe Diao, Ximing Lu, Mingjie Liu, Jiacheng Xu, Xin Dong, Yonggan Fu, Peter Belcak, Hanrong Ye, Hongxu Yin, Yi Dong, Evelina Bakhturina, Tao Yu, Yejin Choi, Jan Kautz, and Pavlo Molchanov. Toolorchestra: Elevating intelligence via efficient model and tool orchestration. *arXiv preprint arXiv:2511.21689*, 2025.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, 2026.

Ke Wang, Houxing Ren, Aojun Zhou, Zimu Lu, Sichun Luo, Weikang Shi, Renrui Zhang, Linqi Song, Mingjie Zhan, and Hongsheng Li. Mathcoder: Seamless code integration in llms for enhanced mathematical reasoning. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

Peng-Yuan Wang, Tian-Shuo Liu, Chenyang Wang, Yi-Di Wang, Shu Yan, Cheng-Xing Jia, Xu-Hui Liu, Xin-Wei Chen, Jia-Cheng Xu, Ziniu Li, and Yang Yu. A survey on large language models for mathematical reasoning. *ACM Computing Surveys*, 2025a.

Wenxiao Wang, Priyatham Kattakinda, and Soheil Feizi. Maestro: Joint graph & config optimization for reliable ai agents. *arXiv preprint arXiv:2509.04642*, 2025b.

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

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2025c.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

Yangbo Wei, Zhen Huang, Lei He, Li Huang, Ting-Jung Lin, and Wei W. Xing. Vflow: Discovering optimal agentic workflows for verilog generation. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2026.

Rong Wu, Xiaoman Wang, Jianbiao Mei, Pinlong Cai, Daocheng Fu, Cheng Yang, Licheng Wen, Xuemeng Yang, Yufan Shen, Yuxin Wang, and Botian Shi. Evolver: Self-evolving llm agents through an experience-driven lifecycle. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2026.

Huajian Xin, Daya Guo, Zhihong Shao, Z.Z. Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. Advancing theorem proving in llms through large-scale synthetic data. *The 4th Workshop on Mathematical Reasoning and AI (MATH-AI) at NeurIPS*, 2024.

Huajian Xin, Z.Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, Wenjun Gao, Haowei Zhang, Qihao Zhu, Dejian Yang, Zhibin Gou, Z.F. Wu, Fuli Luo, and Chong Ruan. Deepseek-prover-v1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025.

Sikuan Yan, Xiufeng Yang, Zuchao Huang, Ercong Nie, Zifeng Ding, Zonggen Li, Xiaowen Ma, Jinhe Bi, Kristian Kersting, Jeff Z. Pan, Hinrich Schuetze, Volker Tresp, and Yunpu Ma. Memory-r1: Enhancing large language model agents to manage and utilize memories via reinforcement learning. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2026.

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, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 Technical Report. *arXiv preprint arXiv:2505.09388*, 2025.

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. Satlm: Satisfiability-aided language models using declarative prompting. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. Aflow: Automating agentic workflow generation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025a.

Mingda Zhang, Wenjin Liu, Tiesunlong Shen, Qika Lin, Rui Mao, Erik Cambria, Xiaoying Tang, and Haoran Luo. Flowsteer: Towards agents designing agentic workflows via reinforced progressive canvas editing. *arXiv preprint arXiv:2602.01664*, 2026.

Xiaoqing Zhang, Yuhan Liu, Flood Sung, Xiuying Chen, Shuo Shang, and Rui Yan. Thinking before running! efficient code generation with thorough exploration and optimal refinement. In *Findings of the Association for Computational Linguistics (ACL)*, 2025b.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the Association for the Advancement of Artificial Intelligence (AAAI)*, 2024.

Mingming Zhao, Xiaokang Wei, Yuanqi Shao, Kaiwen Zhou, Lin Yang, Siwei Rao, Junhui Zhan, and Zhitang Chen. A2flow: Automating agentic workflow generation via self-adaptive abstraction operators. In *Proceedings of the Association for the Advancement of Artificial Intelligence (AAAI)*, 2026.

Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step-by-step. In *Findings of the Association for Computational Linguistics (ACL)*, 2024.

Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Sercan Ö. Arık. Multi-agent design: Optimizing agents with better prompts and topologies. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2026.

Table 3: Implementation details grouped by method category. All methods use Qwen3-Coder-30B-A3B-Instruct as the frozen task model with thinkingmode disabled. Learning-based methods additionally train a Qwen3-4B policy, while CROW further uses Qwen3-Embedding-0.6B for memory retrieval.

Category	Method	Key Hyperparameters
Base	Text	temperature=0, top_p=1.0, max_tokens=2048
	Code	temperature=0, top_p=1.0, max_tokens=2048
Test-Time	Plan-and-Solve	temperature=0, calls=1, top_p=1.0, max_tokens=2048
	Self-Refine	init_temperature=0, refine_temperature=0.7, max_iters=4, top_p=1.0, max_tokens=2048
	Multi-Agent Debate	temperature=1.0, rounds=2, top_p=1.0, max_tokens=2048
Search-Based Workflow	AFlow	rounds=20, optimizer_temperature=1.0, optimizer_max_tokens=8192, workflow_temperature=0, workflow_max_tokens=4096
	ADAS	init_archs=7, generations=30, meta_temperature=0.8, meta_max_tokens=4096, agent_temperature=0.8/0.1, repair_attempts=5
	MASS	mipro_trials=2, max_demos=2, max_agents=10, topology_temperature=0.05, task_temperature=0.7, max_tokens=4096
	EvoMAS	p_mutation=0.8, p_crossover=0.2, meta_temperature=0.7, meta_max_tokens=8192, worker_temperature=0.7, worker_max_tokens=4096, aggregator_temperature=0
Learning-Based Workflow	Prompt-R1	LR=1×10^{-5}, rollouts=4, epochs=1, r=32, α=64, max_steps=8, policy_train_top_p=0.95, policy_train_action_tokens=512, policy_eval_action_tokens=192, expert_max_tokens=2048
	AgentFlow	LR=1×10^{-6}, rollouts=4, batch=32, epochs=1, max_tool_steps=3, train_top_p=0.95, eval_max_tokens=2048
	The Conductor	LR=1×10^{-5}, rollouts=4, batch=32, epochs=1, r=32, α=64, max_workflow_steps=5, train_top_p=0.95, conductor_eval_max_tokens=1024, worker_max_tokens=2048
	CROW	LR=1×10^{-5}, rollouts=4, batch=32, epochs=1, r=32, α=64, max_steps=12, train_temperature=1.0, train_top_p=0.95, max_action_tokens=192, solver_max_tokens=2048, retrieval=top-1

A Baseline Implementation and Adaptation Details

Common experimental setup. Unless otherwise stated, all baselines use Qwen3-Coder-30B-A3B-Instruct as the task-solving model with thinking mode disabled.

We distinguish between inference-only, training-based, and system-optimization baselines. Inference-only methods do not access any training or optimization examples and are applied directly to the held-out test set using fixed inference procedures. Training-based methods explicitly update model parameters using their prescribed training data. Methods that perform workflow search, architecture search, evolutionary optimization, or prompt/topology optimization keep the task model frozen, but require a separate set of problems on which candidate systems can be evaluated during optimization.

For all such system-optimization methods, we use the same set of 100 optimization problems, sampled approximately evenly across the constituent benchmark sources. We intentionally restrict system-level optimization to this subset because its computational cost is substantially larger than ordinary model inference. Unlike a single forward pass, these methods repeatedly construct and evaluate candidate multi-agent systems over many optimization rounds. Each candidate may itself contain several agents and multiple sequential or parallel LLM calls, and every candidate must be executed over the optimization set to obtain a fitness signal. Consequently, the total cost grows multiplicatively with the number of optimization examples, candidate systems, search generations, agents per system, and interaction rounds. Using the full available training corpus would therefore require an impractically large number of long-context Qwen3-Coder-30B generations and would

make the experiment primarily a comparison of available search compute rather than optimization strategy.

The fixed 100-example set provides a common and computationally tractable search budget while retaining coverage across all benchmark sources. It is sampled once independently of model predictions, and is shared by AFlow, ADAS, EvoMAS, and MASS. After optimization, only the selected workflow or agent system is retained, frozen, and evaluated on the complete 1,394-example held-out test set. The exact frozen architectures selected by these searches are reproduced in Appendix B.

All final predictions are evaluated using the same SymbolicAutopoiesis evaluator, including deterministic answer normalization and GPT-4o judging at temperature 0 for cases requiring semantic or mathematical equivalence assessment. Held-out test labels are never exposed during reasoning, refinement, workflow selection, architecture generation, evolutionary search, prompt optimization, or early stopping.

A.1 TEST-TIME

These methods use no training or optimization examples. They operate directly at test time using fixed prompting or interaction procedures and do not produce a persistent learned or searched system.

Direct prompting. Baseline. The baseline queries Qwen3-Coder-30B-A3B-Instruct once per problem. The prompt asks the model to reason step by step and terminate with exactly one FINAL ANSWER: field. Multiple-choice problems are rendered with lettered options, and the model must return the corresponding option label rather than its numeric index. This baseline performs no training, retrieval, self-correction, or multi-agent communication.

Plan-and-Solve. We implement the Plan-and-Solve prompting strategy as a single-pass inference baseline. For each problem, Qwen3-Coder-30B is prompted to first identify the relevant variables, constraints, and mathematical principles, then formulate a complete step-by-step solution plan, execute the plan with explicit intermediate calculations, and finally verify the resulting solution for calculation errors, omitted steps, inconsistent assumptions, unit mismatches, and answer-formatting issues. Both planning and solution generation are performed within a single model call, without any additional refinement or verification stage. We retain the benchmark-specific answer-formatting instructions used in the baseline.

Self-Refine. The Self-Refine baseline first produces one deterministic solution and then performs iterative feedback-and-revision with the same model. At every iteration, the model receives the original problem, its current response, and the preceding refinement history. It must produce specific feedback, a CORRECT or NEEDS REVISION decision, and a complete rewritten solution. Refinement terminates when the model declares the solution correct or when the iteration budget is exhausted. The reference answer is never supplied to the model and is not used for early stopping. We retry refinement responses that do not follow the required output structure.

Multi-agent debate. Our multi-agent debate implementation uses independent instances of. In the first round, all agents solve the problem independently. In the second round, each agent receives the other agents' previous solutions and is instructed to inspect their reasoning, identify possible errors, and produce an updated solution. The final prediction is selected by an unweighted majority vote over the second-round answers after canonical normalization. When no strict majority exists, ties are resolved deterministically according to first occurrence.

A.2 SEARCH-BASED WORKFLOW

These methods keep the underlying language model frozen but search for a persistent workflow or multi-agent architecture using the shared 100-example optimization set. Once search is complete, the selected system is frozen before evaluation on the held-out test set.

AFlow: workflow search. AFlow searches over executable mathematical reasoning workflows. We adapt its graph optimizer and evaluator to the SymbolicAutopoiesis input and answer formats

while retaining its iterative workflow-revision procedure. Search runs for a fixed number a fixed number of rounds without convergence-based early stopping. At every round, the current workflow is evaluated once on all 100 optimization examples. The optimizer samples from the highest-scoring prior rounds when constructing its search context.

After the final round, we select the workflow with the highest optimization accuracy and evaluate the frozen workflow once on the complete held-out test set. In the reported run, the selected workflow was produced at round 13.

ADAS: architecture synthesis. We adapt ADAS Meta-Agent Search to synthesize executable multi-agent programs for symbolic mathematics. Search begins from the method's predefined architectures. Each initial architecture is evaluated on all 100 optimization problems, after which the meta-agent proposes one new architect. Each architecture is evaluated once, and the archive retains all candidates together with their observed fitness. The meta-agent receives a compact summary of the full archive together with full implementations of high-performing and recent candidates. Invalid generated programs may undergo error-driven repair attempts. We select the architecture with the highest mean optimization fitness from the complete archive and evaluate only this frozen winner on the held-out test set.

MASS: joint prompt and topology optimization. We implement the three-stage mathematical variant of Multi-Agent System Search. In Stage 1, block-level prompt optimization uses DSPy MIPROv2 to separately optimize the instructions and demonstrations of a base predictor and the minimum reflection, debate, and aggregation blocks. Block influence is estimated by dividing each block's optimization accuracy by the accuracy of the optimized base predictor.

In Stage 2, topology optimization uses these block-influence scores to define a low-temperature rejection-sampling distribution over the number of aggregation branches, reflection rounds, and debate rounds. We sample and evaluate two unique workflow topologies, with each workflow restricted to at most ten logical agents.

In Stage 3, workflow-level prompt optimization jointly re-optimizes the prompts and demonstrations of all active modules in the selected topology. This allows individual module instructions to be adapted to their roles in the final multi-agent workflow rather than optimized only in isolation.

A full MASS search is substantially more expensive than the other system-optimization baselines because it combines repeated prompt optimization with topology search and subsequent workflow-level re-optimization. We therefore use a reduced and explicitly reported search budget: two instruction candidates, two MIPRO trials, at most two bootstrapped demonstrations, two topology candidates, one evaluation repeat, topology-sampling temperature 0.05, and eight parallel workers. The complete three-stage procedure uses the shared 100-example optimization set, after which the final compiled workflow is frozen and evaluated on all 1,394 held-out test examples.

EvoMAS: evolutionary multi-agent optimization. EvoMAS evolves a structured multi-agent configuration from a pool of seed workflows. We use two initial parents and three evolutionary steps. For this run, the selected seed parents are a five-agent debate workflow and a five-agent peer-review workflow. The debate parent contains two initial solvers, two cross-informed revision agents, and one aggregator. The peer-review parent contains two creators, a reviewer, a reviser, and an aggregator. Each parent is first adapted to the symbolic-mathematics task.

The selection reward combines task accuracy and inference cost:

$$R = \text{accuracy} - 10^{-6}\left(\text{tokens} + 10^{3}\text{time}\right).$$

The highest-reward configuration across the initial parents and all generated offspring is frozen and evaluated on the complete held-out test set. Thinking mode is disabled for all OpenAI-compatible Qwen calls.

A.3 Learning-based Workflow

This category contains methods that explicitly update model parameters rather than only increasing test-time computation or searching over a frozen agent systembaseline.

Prompt-R1. Prompt-R1 consists of a trainable prompt policy and a frozen expert. The policy may either issue an interaction prompt to the expert or terminate with an answer. The expert receives the original problem, the preceding policy–expert transcript, and the policy's latest request.

AgentFlow. We adapt AgentFlow by training the planner and using a frozen model for query analysis, execution, verification, final response generation, and tool-backed reasoning. Because the evaluation tasks are entirely mathematical, we enable only the general natural-language generator and Python coding tools while external search tools are disabled.

The Conductor. We adapt The Conductor as a learned workflow generator. A LoRA-adapted policy produces a multi-step workflow for each problem a multi-step workflow for each problem. Each step specifies a worker, a natural-language subtask, and whether the worker may access all preceding step outputs. The policy may invoke this worker repeatedly for independent solving, verification, refinement, or final synthesis. The final worker response is used as the prediction. At evaluation, conductor decoding is greedy while each Qwen3-Coder-30B worker call is deterministic. When the context exceeds this limit, the oldest preceding workflow messages are removed first.

B Selected Search-Based Workflow Architectures

This section reproduces the frozen architectures selected by each search-based workflow baseline. The listings retain the control flow used for held-out evaluation while omitting framework boilerplate unrelated to routing. Thus, they describe the evaluated systems rather than the broader search spaces from which the systems were selected.

B.1 AFlow

AFlow selected round 13 on the optimization split. The workflow generates three independent solutions, applies an LLM-based self-consistency selector, and invokes a programmer to verify the selected solution. As shown below, the verification result is computed but is not used to alter the returned answer.

AFlow Selected Workflow (Round 13)

```
class Workflow:
    def __init__(self, name, llm_config, dataset):
        self.name = name
        self.dataset = dataset
        self.llm = create_llm_instance(llm_config)
        self.custom = operator.Custom(self.llm)
        self.sc_ensemble = operator.ScEnsemble(self.llm)
        self.programmer = operator.Programmer(self.llm)

    async def __call__(self, problem):
        solutions = []
        for _ in range(3):
            solution = await self.custom(
                input=problem,
                instruction=LIMIT_EVALUATION_PROMPT,
            )
            solutions.append(solution["response"])

        best_solution = await self.sc_ensemble(
            solutions=solutions,
            problem=problem,
        )
        verification = await self.programmer(
```

```
            problem=problem,
            analysis=best_solution["response"],
        )
        return best_solution["response"], self.llm.
            get_usage_summary()["total_cost"]
```

B.2 ADAS

ADAS selected architecture 34 from generation 28, named *Robust Mathematical Reasoning Synthesizer*. It obtains solutions from four role-specialized agents, scores every solution, iteratively refines the two highest-scoring candidates, and returns the first candidate accepted by a validator. If this process fails, it falls back to majority voting.

ADAS Selected Architecture (Architecture 34)

```
def forward(self, taskInfo):
    roles = [
        "Symbolic Mathematician",
        "Calculus Specialist",
        "ODE/PDE Specialist",
        "General Mathematics Expert",
    ]
    experts = [
        LLMAgentBase(["thinking", "answer"], "Expert Agent",
            role=role)
        for role in roles
    ]

    expert_solutions = []
    for expert in experts:
        thinking, answer = expert(
            [taskInfo],
            "Solve this mathematics problem thoroughly with
                rigorous "
            "mathematical reasoning. Return the exact final
                answer.",
        )
        expert_solutions.append(answer)

    assessor = LLMAgentBase(
        ["correctness_score", "completeness_score", "
            clarity_score"],
        "Quality Assessor",
        temperature=0.1,
    )
    scored_solutions = []
    for i, solution in enumerate(expert_solutions):
        try:
            correctness, completeness, clarity = assessor(
                [taskInfo, solution],
                "Evaluate correctness (0-10), completeness
                    (0-10), "
                "and clarity (0-10).",
                i,
            )
```

```
            score = (
                0.40 * int(str(correctness.content).strip())
                + 0.35 * int(str(completeness.content).strip
                    ())
                + 0.25 * int(str(clarity.content).strip())
            )
            scored_solutions.append((score, solution))
        except Exception:
            continue

    if scored_solutions:
        scored_solutions.sort(key=lambda x: x[0], reverse=
            True)
        refined_solutions = []
        for score, solution in scored_solutions[:2]:
            current_solution = solution
            for iteration in range(3):
                try:
                    critic = LLMAgentBase(
                        ["refinement_feedback", "
                            needs_improvement"],
                        "Refinement Critic",
                        temperature=0.2,
                    )
                    feedback, needs_improvement = critic(
                        [taskInfo, current_solution],
                        "Analyze this solution for errors or
                            logical gaps. "
                        "Return TRUE if improvements are
                            needed, FALSE otherwise.",
                        iteration,
                    )
                    if "false" in str(needs_improvement.
                        content).lower():
                        break
                    refiner = LLMAgentBase(
                        ["revised_thinking", "revised_answer
                            "],
                        "Solution Refiner",
                        temperature=0.1,
                    )
                    _, current_solution = refiner(
                        [taskInfo, current_solution, feedback
                            ],
                        "Revise the solution completely based
                             on the feedback.",
                        iteration,
                    )
                except Exception:
                    break
            refined_solutions.append(current_solution)

        validator = LLMAgentBase(
            ["validation_status"], "Final Validator",
                temperature=0.05
        )
        try:
```

```
            for solution in refined_solutions:
                validation = validator(
                    [taskInfo, solution],
                    "Verify this is a mathematically correct
                       and complete solution.",
                    0,
                )
                status = str(validation.content).lower()
                if "valid" in status or "correct" in status:
                    return solution
        except Exception:
            pass

    return Info(
        "answer",
        "Robust Mathematical Reasoning Synthesizer",
        majority_vote([str(x.content) for x in
           expert_solutions]),
        iteration_idx=-1,
    )
```

B.3 MASS

MASS selected one aggregation branch, two reflection rounds, and two debate rounds. Because debate is active, the branch begins with two independently predicted solutions. Each may undergo two reflector–refiner cycles, followed by two debate updates. With one aggregation branch, the final vote operates over a single branch output.

MASS Selected Topology and Execution

```
selected_topology = WorkflowConfig(
    aggregate=1,
    reflect=2,
    debate=2,
)

class MassWorkflow(dspy.Module):
    def _reflect(self, question, answer):
        current = answer
        for _ in range(self.config.reflect):
            review = self.reflector(
                question=question,
                text=str(current.answer),
            )
            if parse_correctness(review.correctness):
                break
            current = self.refiner(
                question=question,
                previous_answer=str(current.answer),
                reflection=review.feedback,
                correctness=str(review.correctness),
            )
        return current

    def _chain(self, question):
```

```
        predictor_count = 2 if self.config.debate else 1
        history = [
            self._reflect(question, self.predictor(question=
                question))
            for _ in range(predictor_count)
        ]
        current = history[0]
        for _ in range(self.config.debate):
            solutions = "\n\n".join(
                f"Agent {i}: {item.answer}"
                for i, item in enumerate(history)
            )
            current = self.debator(
                question=question,
                solutions=solutions,
            )
            history.append(current)
        return current

    def forward(self, question):
        branches = [
            self._chain(question)
            for _ in range(self.config.aggregate)
        ]
        answers = [extract_answer(str(x.answer)) for x in
            branches]
        counts = Counter(normalize_math(x) for x in answers
            if x.strip())
        normalized = counts.most_common(1)[0][0]
        winner = next(x for x in answers if normalize_math(x)
            == normalized)
        return dspy.Prediction(answer=winner, branch_answers=
            answers)
```

B.4 EvoMAS

EvoMAS selected a mutated peer-review configuration named `symbolic_peer_review`. Two creators execute in parallel, their outputs are passed to a reviewer, the review is passed to a reviser, and the result is passed to an aggregator. Every node is a Qwen3-Coder-30B CodeAgent without additional tools.

EvoMAS Selected Multi-Agent Configuration

```
name: symbolic_peer_review
backend: smolagents

agents:
  creator1:
    role: creator
    agent_type: CodeAgent
    model_id: openai:qwen3-coder-30b-a3b-instruct
    prompt: reviewer_create
    tools: []
    max_tokens: 8192
    temperature: 0.7
```

```
    creator2:
      role: creator
      agent_type: CodeAgent
      model_id: openai:qwen3-coder-30b-a3b-instruct
      prompt: reviewer_create
      tools: []
      max_tokens: 8192
      temperature: 0.7
    reviewer:
      role: reviewer
      agent_type: CodeAgent
      model_id: openai:qwen3-coder-30b-a3b-instruct
      prompt: mathematical_review
      tools: []
      max_tokens: 8192
      temperature: 0.7
    reviser:
      role: reviser
      agent_type: CodeAgent
      model_id: openai:qwen3-coder-30b-a3b-instruct
      prompt: reconstruct_solution_from_review
      tools: []
      max_tokens: 8192
      temperature: 0.8
    aggregator:
      role: aggregator
      agent_type: CodeAgent
      model_id: openai:qwen3-coder-30b-a3b-instruct
      prompt: aggregator
      tools: []
      max_tokens: 8192
      temperature: 0.5

  topology:
    reports_to:
      creator1: [reviewer]
      creator2: [reviewer]
      reviewer: [reviser]
      reviser: [aggregator]

  execution:
    parallel_workers: true
    timeout: 600
    max_retries: 3
```

C INFERENCE-COST ESTIMATION

We estimate inference cost on a fixed stratified sample of 50 held-out questions, containing 10 questions from each benchmark. For each query, a call is one invocation of Qwen3-Coder-30B, and token cost is the sum of its input and output tokens across all such invocations. We then average both quantities over the 50 queries. Calls to the Qwen3-4B policy or controller, embedding retrieval, judging, and local program execution are excluded so that the table isolates use of the shared task-solving backbone.

For AFlow, ADAS, and MASS, we rerun the frozen selected workflow and sum the token usage returned by the inference server for every call. The Conductor is measured from its recorded runtime

Table 4: Estimated Qwen3-Coder-30B inference cost per query.

Method	Calls	Tokens
Plan-and-Solve	1.00	1,679.64
Self-Refine	3.88	10,166.42
Multi-Agent Debate	6.00	21,501.14
AFlow	5.00	12,441.76
ADAS	21.32	20,075.12
MASS	7.30	13,924.80
EvoMAS	29.63	184,175.22
Prompt-R1	1.08	1,415.92
AgentFlow	6.98	15,708.18
The Conductor	4.70	8,453.64
CROW	1.96	7,306.44

usage. For Plan-and-Solve, Self-Refine, Multi-Agent Debate, Prompt-R1, AgentFlow, and CROW, we reconstruct the Qwen3-Coder-30B calls from their saved per-query outputs or trajectories and tokenize the corresponding prompts and responses with the backbone tokenizer when server usage is unavailable. For EvoMAS, whose CodeAgents can make several internal turns, we reconstruct every recorded internal model turn from its saved traces; partial turns completed before a timeout remain part of the query cost. Thus, the reported values are directly recorded where usage metadata exists and tokenizer-based approximations otherwise, under the same call and token scope for every method.

D Recurring Baseline Failure

This section compares non-base baselines on the same HARDMath problem used in the complete CROW trajectory in Appendix J.1. For $a_1x^{10} + a_2x^6 + a_3 \mapsto \epsilon y^{10} + y^6 + 1$, the reference answer is $\epsilon = (a_1/a_3)(a_3/a_2)^{5/3} = a_1a_3^{2/3}/a_2^{5/3}$.

We report every intermediate output preserved by each evaluation harness. Complete stage-level traces are available for Plan-and-Solve, Self-Refine, Multi-Agent Debate, EvoMAS, Prompt-R1, AgentFlow, and The Conductor. AFlow retains only its selected response, while ADAS and MASS retain only the final held-out return and external judge result. We mark these three cases as output-only evidence rather than presenting known architecture stages as if their intermediate messages had been observed.

D.1 Plan-and-Solve

Plan-and-Solve Persisted Trajectory

Plan and solution:
Let $x = ky$ and match
$a_1 k^{10} = \epsilon$, $a_2 k^6 = 1$, and
$a_3 = 1$.
Thus $k = a_2^{-1/6}$ and
$k^{10} = a_2^{-5/3}$.

The response notices that the problem does not specify $a_3 = 1$, but
repeats the same coefficient matching instead of normalizing the polynomial.

RETURNED: $\epsilon = a_1 / a_2^{5/3}$

D.2 SELF-REFINE

Self-Refine Persisted Trajectory

Initial answer:
$\epsilon = a_1 / a_2^{5/3}$

Revision 1 feedback:
Do not impose $a_3 = 1$; account for the constant normalization.
Revision 1 answer:
$\epsilon = a_1 / a_2^{5/3}$

Revision 2 feedback:
Normalize the complete expression using a_3.
Revision 2 answer:
$\epsilon = a_1 / a_2^{5/3}$

Revision 3 feedback:
Divide by a_3 before applying the variable scale.
Revision 3 derivation begins using
$\ell^6 = a_3 / a_2$.

Revision 4 derivation:
$\ell^6 = a_3 / a_2$,
$\epsilon = (a_1 / a_3) \ell^{10}$, and
$\ell^{10} = (a_3 / a_2)^{5/3}$.

Revision 4 then makes an exponent-sign error and returns:
$\epsilon = a_1 / (a_2^{5/3} a_3^{2/3})$

D.3 MULTI-AGENT DEBATE

Multi-Agent Debate Persisted Trajectory

Round 1:
Agent 1: $\epsilon = a_1/a_2^{5/3}$
Agent 2: $\epsilon = a_1/a_2^{5/3}$
Agent 3: $\epsilon = a_1/a_2^{5/3}$

Round 2 after each agent receives its peers' responses:
Agent 1: $\epsilon = a_1/a_2^{5/3}$
Agent 2: $\epsilon = a_1/a_2^{5/3}$
Agent 3: $\epsilon = a_1/a_2^{5/3}$

Vote:
$a_1/a_2^{5/3}$ receives 3/3 votes.

RETURNED: $\epsilon = a_1/a_2^{5/3}$

Repeated sampling increases agreement rather than correcting the shared error.

D.4 AFLOW

AFlow Persisted Selected Response

Let $x = ky$ and match:
$a_1 k^{10} = \epsilon$,
$a_2 k^6 = 1$, and
$a_3 = 1$.

Therefore $k = a_2^{-1/6}$ and
$\epsilon = a_1/a_2^{5/3}$.

RETURNED: $\epsilon = a_1/a_2^{5/3}$

AFlow's selected architecture generated three candidates, selected one by self-consistency, and ran a programmer check. Its held-out artifact stores only the selected response above, not the three candidates or checker output. Accordingly, this is final selected-response evidence, not a complete internal trajectory.

D.5 ADAS

ADAS Persisted Held-Out Record

Architecture:
Robust Mathematical Reasoning Synthesizer (architecture 34)

RETURNED: $\epsilon = a_1/a_2^{5/3}$

External judge:
Incorrect; the answer omits $a_3^{2/3}$.

The ADAS harness stores the final return and judge result, but not the four expert proposals, scores, refinements, or validator message for this held-out instance. This record proves what ADAS finally committed to, but cannot identify which internal agent introduced or endorsed the error.

D.6 MASS

MASS Persisted Held-Out Record

Selected topology:
aggregate=1, reflect=2, debate=2

RETURNED: $\epsilon = a_1/a_2^{5/3}$

External judge:
Incorrect; the answer omits $a_3^{2/3}$.

The MASS harness likewise stores the final branch answer and judge result, but not the initial predictions, reflection feedback, refined answers, or debate messages. This is final-output evidence rather than a reconstructed trajectory.

D.7 EvoMAS

EvoMAS Complete Persisted Agent Trace

Creator 1 output:
$a_1/a_2^{1.66666666666667}$

Creator 2 output:
$\epsilon = a_1/a_2^{5/3}$

Reviewer output:
$a_1/a_2^{5/3}$

Reviser output:
"Based on the mathematical derivation and confirmed by the prior agents' work, the answer is $\epsilon = a_1/a_2^{5/3}$. This is the expression for epsilon as a function of a_1, a_2, a_3 that nondimensionalizes the polynomial."

Aggregator output:
$a_1/a_2^{5/3}$

Both creators independently omit a_3, the reviewer endorses the same answer, the reviser treats agreement as confirmation, and the aggregator returns it unchanged. The full multi-agent path amplifies rather than detects the shared error.

D.8 PROMPT-R1

Prompt-R1 Complete Persisted Trace

Policy output 1 -- INTERACTION:
"Let $y = x^k$. Matching x^{10} to y^{10} gives $k = 1$, hence $y = x$. Comparing coefficients gives $\epsilon = a_1$."

Weakness exposed: the learned policy prompt already commits to an incorrect transformation before the expert begins reasoning.

Expert output:
"With $y = x$, compare $a_1 y^{10} + a_2 y^6 + a_3$ with $\epsilon y^{10} + y^6 + 1$. Thus $\epsilon = a_1$, $a_2 = 1$, and $a_3 = 1$."

The expert follows and elaborates the anchored premise rather than reconstructing the missing scaling relation.

Policy output 2 -- ANSWER:
$\epsilon = a_1$

D.9 AGENTFLOW

AgentFlow Complete Persisted Trace

Step 1 -- action:
Call Generalist Solution Generator for a symbolic expression.

Step 1 -- tool output:
$x = ky$, $a_2 k^6 = 1$, $a_3 = 1$, and $\epsilon = a_1 / a_2^{5/3}$.

Step 1 -- verifier:
CONTINUE. The result is incomplete because it assumes $a_3 = 1$.

The verifier correctly detects thereasoning defect.

Step 2 -- tool output:
Repeats $\epsilon = a_1 / a_2^{5/3}$ and notes the unresolved requirement $a_3 = 1$.

Step 2 -- verifier:
CONTINUE. Further validation is required to include a_3.

Weakness exposed: despite explicit feedback, the same generator preserves the omitted normalization in its next attempt.

Step 3 -- tool output:

Again derives $\epsilon = a_1/a_2^{5/3}$.

Final synthesis:
Claims the constant normalization is captured by
$\epsilon = a_1/a_3$.

RETURNED: $\epsilon = a_1/a_3$

AgentFlow's verifiers identify the omission twice, but repeated calls to the same generator do not repair it; synthesis introduces a different incomplete ratio.

D.10 The Conductor

The Conductor Complete Persisted Trace

Worker 1 -- solve (independent):
Sets $x = ky$, matches $a_3 = 1$, and returns
$\epsilon = a_1/a_2^{5/3}$.

Worker 2 -- determine scaling (independent):
Repeats the same coefficient matching and returns
$\epsilon = a_1/a_2^{5/3}$.

Both upstream workers establish the same flawed premise before verification begins.

Worker 3 -- verify (access to all prior outputs):
$\boxed{\epsilon = a_1/a_2^{5/3}}$

Worker 4 -- refine (access to all prior outputs):
$\boxed{\epsilon = a_1/a_2^{5/3}}$

Worker 5 -- format (access to all prior outputs):
$\boxed{\epsilon = a_1/a_2^{5/3}}$

Weakness exposed: verification, refinement, andformatting all inherit and reinforce the shared upstream error.

RETURNED: $\epsilon = a_1/a_2^{5/3}$

Verification and refinement repeat the initial answer because their context contains two mutually reinforcing versions of the same mistake.

D.11 Recurring Failure and CROW's Correction

The complete traces for Plan-and-Solve, Self-Refine, Multi-Agent Debate, EvoMAS, Prompt-R1, AgentFlow, and The Conductor show the same failure surviving planning, feedback, debate, review, verification, and learned decomposition. The final recorded outputs of AFlow, ADAS, and MASS exhibit the same omitted factor, although their harnesses do not preserve enough intermediate messages to attribute the error to a particular internal stage. Across methods, the local substitution (*@$x = ky$@*) omits the output scale required to normalize (*@a_3@*). CROW instead retrieves

the coupled relations $a_1 = \epsilon L^{10} M,\ a_2 = L^6 M,\ a_3 = M$. As Appendix J.1 shows, these imply $M = a_3$, $L^6 = a_2/a_3$, and the correct $\epsilon = a_1 a_3^{2/3}/a_2^{5/3}$.

E Ablation Implementation Details

E.1 Shared Components

All ablations use Qwen3-4B as the orchestrator and keep the Qwen3-Coder-30B-A3B-Instruct task-solving backbone fixed. They share the same natural-language reasoner, programmer, modality-specific revisers, program executor, answer-submission protocol, and evaluator. Consequently, removing constraints or memory changes the information and actions available to the orchestrator, rather than replacing the downstream solvers. The six variants use three orchestration prompt families: the direct-only prompt for Ablations 5 and 6, the constraint-aware but memory-free prompt for Ablations 3 and 4, and the full adaptive prompt for Ablation 2 and CROW. The complete prompt families are provided in Appendix F.

E.2 Ablation 6: No Constraint, No Memory, No RL

This variant uses the direct-only orchestration protocol with the base, untrained Qwen3-4B orchestrator. For each problem, the orchestrator selects either the natural-language reasoner or the programmer. It may subsequently invoke the corresponding reviser to repair a concrete defect before submitting the answer. Constraint construction, constraint-use fields, retrieval, and memory access are unavailable.

E.3 Ablation 5: No Constraint, No Memory, with RL

This variant uses exactly the same direct-only prompt, action space, downstream operators, and runtime as Ablation 6. The only component change is that the Qwen3-4B orchestrator is optimized with GRPO using downstream answer correctness as the reward. It therefore isolates the effect of learning how to select between direct natural-language reasoning, programming, revision, and answer submission.

E.4 Ablation 3: Constraint, No Memory, No RL

This variant augments the workflow with constraint construction while retaining the base Qwen3-4B orchestrator. Before calling a solver, the orchestrator must judge whether a critical constraint would materially help. A Need-Constraint decision invokes the Constraint Provider, which produces either natural-language verification guidance or an executable verifier and routes the result to the corresponding solver. A No-Constraint decision permits direct natural-language or programmatic solving. No retrieval or memory action is available.

E.5 Ablation 4: Constraint, No Memory, with RL

This variant uses exactly the same constraint-aware, memory-free prompt, action space, and operators as Ablation 3, but replaces the base orchestrator with the GRPO-trained Qwen3-4B policy. It therefore measures whether policy optimization alone can learn when to construct a constraint, which solver modality to use, and when a failed output should be revised.

E.6 Ablation 2: Constraint, Memory, No RL

This variant exposes the complete adaptive routing protocol to the base Qwen3-4B orchestrator. The orchestrator first evaluates whether the retrieved problem is structurally relevant. A matching entry may be opened and its constraint applied; otherwise, the orchestrator decides between proposing a temporary constraint and solving directly. Retrieval uses the top-ranked entry from the memory snapshot produced during CROW training. This snapshot is frozen throughout held-out evaluation: evaluation examples neither modify nor extend the memory.

E.7 Full CROW: Constraint, Memory, with RL

Full CROW combines the complete adaptive protocol with the GRPO-trained Qwen3-4B orchestrator. The policy can reuse a structurally matched memory, construct a new constraint when retrieval is insufficient, solve directly, or invoke the matching reviser after a recoverable failure. During training, a newly proposed constraint used in a correct trajectory may be generalized and promoted to the synchronized memory. The resulting memory snapshot is frozen for final evaluation.

E.8 Constraint-Aware, Memory-Free Orchestrator Prompt

The following prompt is shared by Ablations 3 and 4.

Constraint-Aware Memory-Free Orchestrator Prompt

```
You are the orchestrator for a constraint-enabled
   mathematical workflow with no
memory. You control tools but do not solve the mathematics
   yourself. Return
exactly one allowed action per response.

At STATE: start, call JUSTIFY_CONSTRAINT_NEED:

action: JUSTIFY_CONSTRAINT_NEED
prompt:
CONSTRAINT_NEED: <specific analysis of whether a constraint
   materially helps>
VERDICT: NEED_CONSTRAINT | NO_CONSTRAINT

After NEED_CONSTRAINT, call:

action: CALL_CONSTRAINT_PROVIDER
with_code: true | false
prompt:
<instruction grounded in the accepted analysis>

After NO_CONSTRAINT, call either solver with use_constraint:
   false. After a
constraint is created, solver routing is mandatory:
- with_code: true -> CALL_PROGRAMMER with use_constraint:
   true
- with_code: false -> CALL_NATURAL_LANGUAGE with
   use_constraint: true

Do not send executable verifier code to the natural-language
   solver, and do not
send a natural-language-only constraint to the programmer.

action: CALL_NATURAL_LANGUAGE | CALL_PROGRAMMER
use_constraint: true | false
prompt:
<short solving instruction>

action: CALL_NATURAL_LANGUAGE_REVISER |
   CALL_PROGRAMMER_REVISER
use_constraint: true | false
prompt:
<specific defect and requested repair>
```

```
action: SAVE_ANSWER
prompt:
<copy the exact final answer from the latest successful
    solver result>

Memory actions do not exist. Never request, retrieve, or
    refer to memory.
```

E.9 Direct-Only Orchestrator Prompt

The following prompt is shared by Ablations 5 and 6.

Direct-Only Orchestrator Prompt

```
You are the orchestrator for a plain mathematical problem-
    solving workflow. You
control tools but do not solve the mathematics yourself.
    Return exactly one
allowed action per response.

Available actions and exact formats:

action: CALL_NATURAL_LANGUAGE | CALL_PROGRAMMER
prompt:
<short solving instruction>

action: CALL_NATURAL_LANGUAGE_REVISER |
    CALL_PROGRAMMER_REVISER
prompt:
<specific defect and requested repair>

action: SAVE_ANSWER
prompt:
<copy the exact final answer from the latest successful
    solver result>

Constraints and memory do not exist. Do not request them and
    do not emit a
use_constraint field. Use a reviser only for a concrete
    defect.
```

F Prompt Templates

F.1 Orchestrator

Prompt for Orchestrator

```
You are the orchestrator for a mathematical problem-solving
    workflow. You control
tools, but you do not solve the mathematics yourself. Choose
    exactly one allowed
```

```
action per response.

At STATE: start, your first and only allowed action is
    JUSTIFY_MEMORY. Compare only
the ORIGINAL problem and the retrieved Related question.
    Finish with VERDICT: MATCH
when their mathematical structures match, or VERDICT:
    NO_MATCH otherwise.

If memory does not match, the runtime will ask you to call
    JUSTIFY_CONSTRAINT_NEED.
This is a separate judgment about whether the ORIGINAL
    problem needs a newly
generated constraint or can be solved directly. Do not mix
    this difficulty judgment
into JUSTIFY_MEMORY.

ACTION FORMATS

action: JUSTIFY_MEMORY
prompt:
STRUCTURAL_COMPARISON: <specific comparison of the two
    problem structures>
VERDICT: MATCH | NO_MATCH

action: JUSTIFY_CONSTRAINT_NEED
prompt:
CONSTRAINT_NEED: <specific analysis of whether a constraint
    would materially help>
VERDICT: NEED_CONSTRAINT | NO_CONSTRAINT

action: READ_MEMORY
memory_id: <retrieved memory ID>
prompt:
<brief reason for activating the structurally matched memory>

action: CALL_CONSTRAINT_PROVIDER
with_code: true | false
prompt:
<instruction grounded in the accepted constraint-need
    analysis>

action: CALL_NATURAL_LANGUAGE
use_constraint: true | false
prompt:
<short solving instruction>

action: CALL_PROGRAMMER
use_constraint: true | false
prompt:
<short solving instruction>

action: CALL_NATURAL_LANGUAGE_REVISER
use_constraint: true | false
prompt:
<specific defect and requested repair>
```

```
action: CALL_PROGRAMMER_REVISER
use_constraint: true | false
prompt:
<specific defect and requested repair>

action: SAVE_ANSWER
prompt:
<copy the exact final answer from the latest successful
    solver result>

STATE RULES

- STATE: start: output JUSTIFY_MEMORY with MATCH or NO_MATCH.
- After VERDICT: MATCH: call READ_MEMORY.
- After memory VERDICT: NO_MATCH: call
    JUSTIFY_CONSTRAINT_NEED.
- After VERDICT: NEED_CONSTRAINT: call
    CALL_CONSTRAINT_PROVIDER. Choose with_code:
  true only when executable checking or code-based solving is
      genuinely useful.
- After VERDICT: NO_CONSTRAINT: call a direct solver with
    use_constraint: false.
- STATE: memory_ready or constraint_ready: immediately call
    the solver indicated
  by ACTIVE_VERIFICATION_MODE with use_constraint: true.
- STATE: natural_language_result or program_result:
    SAVE_ANSWER when the result is
  correct and complete. Use the matching reviser only for a
     concrete defect.
- Never invent a memory ID. Memory is frozen and temporary
    constraints are not saved.
- Output exactly one action. Put all action-specific fields
    before prompt:, because
  prompt consumes the remainder of the response.
```

F.2 CONSTRAINT PROVIDER

Prompt for Constraint Provider

```
Identify exactly ONE critical condition that a proposed final
    answer must satisfy.

Do not solve the problem, state the expected answer, or write
    code.

The condition must:
- follow from the original problem;
- directly help distinguish a correct final answer from an
    incorrect one;
- preserve all free variables and choices left open by the
    problem;
- avoid incidental intermediate calculations unless they are
    themselves the
  requested answer;
```

```
- be independently checkable without knowing the answer in
   advance.

A later verifier may recompute a quantity from the original
   problem. Describe the
condition to check, not the value it evaluates to.

Return exactly:

Critical constraint:
<one concise condition>

Verification strategy:
<how to test it independently>

Why this is critical:
<why failure implies the candidate is incorrect>
```

F.3 Constraint Compiler

Prompt for Constraint Compiler

```
Convert the supplied critical constraint into concise
   executable Python verification
code for a candidate answer.

Prefer one verification function:

def verify_<descriptive_name>(candidate, ...):
    ...

The verification should check the candidate against the
   original problem and
independently compute whatever reference quantity is needed.
   Never hard-code or
receive the expected final answer.

Keep the verifier concise.
- Do not include docstrings or explanatory comments.
- Do not restate the mathematical derivation.
- Do not solve the full problem or produce the final answer.
- Avoid unused helpers and duplicated calculations.
- Use only the code necessary to perform the verification.
- Use exact symbolic checks where appropriate and numerical
   tolerances where exact
  equality is inappropriate.

Return executable verification code only.
```

F.4 NATURAL LANGUAGE REASONER

Prompt for Natural Language Reasoner

```
Solve the given mathematical problem using natural-language
    reasoning.

You will be given the original problem and may also be given
    a
[Critical Constraint Analysis] produced by another operator.

If a [Critical Constraint Analysis] is provided, follow these
     steps strictly:

Step 1     Extract:
Write down the relevant equations, relationships, and
    conditions from the
Critical Constraint Analysis exactly as stated. Do not
    reinterpret, replace,
simplify, or derive alternative constraints at this step.

Step 2     Solve:
Start from the extracted constraints and derive the answer
    from them. You MUST
use the extracted relationships. Do not replace them with
    relationships from an
independent solution approach.

Step 3     Verify:
Substitute or check the derived result against the extracted
    constraints. If it
does not satisfy them, revise the solution before answering.

If no [Critical Constraint Analysis] is provided, solve the
    problem normally.

Reason through the problem carefully. Do not write Python
    code or describe a
program.

End with exactly one line in this format:
FINAL ANSWER: <answer>

The complete answer MUST be on the same line as "FINAL ANSWER
    :", and this MUST
be the last non-whitespace line. Do not use "FINAL ANSWER:"
    elsewhere.
Return exactly what the problem asks for.
```

F.5 Natural Language Reviser

Prompt for Natural Language Reviser

```
Revise a previous natural-language solution to the
   mathematical problem.

You will be given:
- the original problem;
- the previous natural-language response;
- an orchestrator revision instruction describing what should
    be reconsidered or fixed;
- when applicable, a protocol failure explaining why the
   previous response was unusable.

Do not start from a blank slate unless the previous approach
   is fundamentally wrong.
Inspect the previous reasoning, preserve correct parts,
   repair incorrect or unsupported
steps, and produce a coherent revised solution.

The original problem is authoritative. Do not invent a
   reference answer.
Do not write Python code and do not describe a program.

If the previous response failed only because its final-answer
    format was malformed,
preserve the mathematical reasoning when it is otherwise
   sound and repair the format.
If the mathematics is wrong or incomplete, repair the
   mathematics as well.

Return the revised reasoning in plain text.

End the response with exactly one final-answer line:

FINAL ANSWER: <answer>

The FINAL ANSWER line must be the last non-whitespace line of
    the response.
Do not use the phrase "FINAL ANSWER:" anywhere else.

Return exactly what the problem asks for. For multiple-choice
    problems, return the
requested option representation. Preserve required
   mathematical notation, units,
variables, and precision.
```

F.6 Programmer

Prompt for Programmer

```
Generate an executable Python program that solves the given
   mathematical problem
and verifies its result.
```

```
You will be given the original problem and may also be given:
- [Critical Constraint Analysis]

The original problem is authoritative.

If [Critical Constraint Analysis] is provided, follow these
    steps:

Step 1     Extract:
Identify the relevant equations, relationships, conditions,
    edge cases, and
verification criteria from the constraint analysis. Preserve
    their mathematical
meaning; do not silently replace them with an unrelated
    solution strategy.

Step 2     Solve:
Write code that derives the requested answer from the problem
     while actively
using the extracted constraints where they apply. Never hard-
    code a known answer.

Step 3     Verify:
Implement whatever programmatic checks are appropriate for
    the extracted
constraints and check the computed candidate before returning
     it. Verification
may be integrated directly into the solving logic or
    implemented with local helper
functions; there is no required verify_* function name or
    externally supplied
verifier. If the candidate violates a relevant valid
    constraint, revise the solving
logic before returning the answer.

If a supplied constraint conflicts with the original problem
    or a valid
mathematical derivation, the original problem remains
    authoritative. Correct or
disregard only the conflicting part.

If no constraint is provided, solve the original problem
    normally and add
lightweight verification when useful.

Return ONLY executable Python source. No Markdown, backticks,
     or prose outside
Python comments. The final non-whitespace line must be
    exactly:

final_answer = solver()

Define a top-level solver() taking no required arguments. It
    must return:

f"final answer: {answer}"
```

```
Derive the answer from the problem in code. Never hard-code a
    known answer.

Keep the program concise.
- Do not explain the derivation in comments.
- Do not restate the problem.
- Do not include unused imports, variables, or helper
   functions.
- Prefer direct SymPy expressions and simple computations
   over elaborate
  procedural code.
- Use only as much code as necessary to compute the answer
   and perform useful
  verification.

Return exactly what the problem asks for. Do not leave
   unresolved intermediate
symbols, unevaluated operators, placeholders, or invented
   free parameters unless
the requested answer explicitly requires them.

Use exact SymPy arithmetic where exactness matters. Do not
   use unresolved symbolic
relations directly as Python booleans.

Allowed imports: Python builtins plus math, cmath, decimal,
   fractions, itertools,
functools, collections, statistics, sympy, numpy, scipy.

Do not use print/input, files, network access, subprocesses,
   environment access,
eval/exec, reflection, private/dunder attributes, or system
   calls.

No statement may follow:

final_answer = solver()
```

F.7 Program Reviser

Prompt for Program Reviser

```
You are a Program Reviser. Your job is to repair a Python
   program that failed during execution.

You will be given the original problem, the previous program,
    the execution result or error, and, when available,
   verification constraints.

Return ONLY the complete revised Python source code. Do not
   use Markdown or include any explanation outside the code.

Fix the actual cause of the failure while preserving the
   intended solution.
```

```
If an assertion or verification check fails, do not simply
   delete, disable, weaken, or bypass it to make the program
    run. Determine whether the solving code or the
   verification code is incorrect, then repair the
   appropriate code.

A verifier is not automatically correct. If it incorrectly
   represents the original problem, fix the verifier rather
   than forcing the solution to satisfy it.

Keep the revision focused. Do not rewrite working parts
   unnecessarily.

The revised program must be self-contained and executable,
   and must end with:

final_answer = solver()
```

F.8 MEMORY CREATOR

Prompt for Memory Creator

```
You are the Memory Creator Operator. You are given the
   constraint analysis and the
verification functions of ONE solving run that executed
   successfully. Your job is
to turn that single run into reusable experience for the
   WHOLE family of problems
it belongs to. You do not solve anything and you do not judge
    whether the run's
answer was right.

Perform exactly three transformations.

1. DE-SPECIALIZE. Remove every problem-specific detail:
   concrete numbers,
   specific coefficients, particular exponents and orders,
      named variables that
   only this problem introduced, the specific interval or
      bound, the particular
   integrand, polynomial, or equation. What survives is the
      ROLE each played.

2. THEORETICAL ABSTRACTION. Rewrite the constraint list as
   the theoretical
   constraint list for this TYPE of problem: the conditions
      any correct answer to
   any instance of this family must satisfy, stated so they
      still have teeth. A
   constraint that has become so vague it could not fail is
      worse than no
   constraint - delete it instead. Keep the answer contract
      and the verification
```

```
    strategy as separate, clearly labelled parts of the same
       list.

3. CODE ABSTRACTION. Rewrite the verification functions as
    ONE generalized
    verification module: parameterized functions that take the
        problem's objects
    as arguments instead of closing over this problem's values
       . Concretely, a check
    like "x must not exceed 5" becomes a bound check that
       takes the bound as a
    parameter; a residual against one specific equation
       becomes a residual check
    taking the equation and the candidate; a check at three
       hardcoded sample points
    becomes a sampling check taking the sample points.
       Preserve the LOGIC of each
    original check, including its tolerance policy - if the
       original accepted an
    asymptotic or numerical match rather than an exact
       identity, the generalized
    version must keep that as a parameter, not harden it into
       an exact test. The
    code must be valid Python 3, use only the standard library
        plus sympy/mpmath/
    numpy, define plain module-level functions, contain no top
       -level executed
    statements, no I/O, and no example call.

KEY NAMING. Produce one short key naming the family and what
    the constraints
govern, in exactly this shape:

     <ProblemType>_<ConstraintRole>

- Both halves are written in CapitalizedWords, multi-word
    halves joined by single
  hyphens, the two halves joined by exactly one underscore.
- ProblemType names the mathematical family, inferred from
    the material. If a
  PROBLEM_TYPE_HINT is supplied, use it, normalized to this
      shape.
- ConstraintRole names what the checks actually enforce, in
    two to four words.
- ASCII letters and digits only, at most 60 characters total,
     no spaces.
- Examples of the SHAPE only, not of what you must output:
  ODE_Check-Assumption, LinearAlgebra_Dimension-Match,
      Series_Truncation-Order.

OUTPUT FORMAT. Return exactly these four lines/blocks and
    nothing else. No
Markdown, no fences, no commentary before or after.

memory_key: <the key>
[GENERALIZED_CONSTRAINTS]
```

```
<the theoretical constraint list, plain text, with the answer
    contract and the
verification strategy as labelled parts>
[GENERALIZED_VERIFICATION_FUNCTION]
<the generalized Python verification module, raw source, no
   fence>
[END]
```

G IMPROVEMENTS WITH CONSTRAINTS

G.1 NONDIMENSIONALIZATION WITH COUPLED SCALING CONSTRAINTS

G.1.1 BASE MODEL FAILURES

The base model exhibits the same coupled-scaling error across two nondimensionalization problems with different polynomial degrees.

Question 1

Nondimensionalize the polynomial $a_1x^7 + a_2x^2 + a_3$ into one of the form $\epsilon y^7 + y^2 + 1$.

Express ϵ as a function of a_1, a_2, and a_3.

Base Model Inference - Question 1

Let me assume there's a scaling relationship of the form $x = ky$ for some constant k to be determined.

Substituting $x = ky$ into the original polynomial:
$a_1(ky)^7 + a_2(ky)^2 + a_3$
$= a_1k^7y^7 + a_2k^2y^2 + a_3$

For this to match the form $\epsilon y^7 + y^2 + 1$, I need:
- The coefficient of y^7 to equal ϵ
- The coefficient of y^2 to equal 1
- The constant term to equal 1

So I have the system:
- $a_1k^7 = \epsilon$
- $a_2k^2 = 1$
- $a_3 = 1$

From equation (3): $a_3 = 1$

From equation (2):
$k^2 = \frac{1}{a_2}$,
so $k = \frac{1}{\sqrt{a_2}}$

Substituting this into equation (1):

$\epsilon = a_1 \left(\frac{1}{\sqrt{a_2}}\right)^7$

Therefore:
$\epsilon = \frac{a_1}{(a_2)^{7/2}}$

FINAL ANSWER: a_1/(a_2**(7/2))

Error: The model treats the target constant coefficient 1 as a restriction $a_3 = 1$ on the original polynomial. It therefore omits the overall scaling needed to normalize the constant term, causing the derived ϵ to lose its dependence on a_3.

Question 2

Nondimensionalize the polynomial $a_1x^9 + a_2x^8 + a_3$ into one of the form $\epsilon y^9 + y^8 + 1$.

Express ϵ as a function of a_1, a_2, and a_3.

Base Model Inference - Question 2

Let me assume there's a substitution of the form $x = ky$ for some constant k.

If I substitute $x = ky$:
$a_1 (ky)^9 + a_2 (ky)^8 + a_3 = a_1 k^9 y^9 + a_2 k^8 y^8 + a_3$

For this to be in the form $\epsilon y^9 + y^8 + 1$, I need:
- The coefficient of y^9 to equal ϵ
- The coefficient of y^8 to equal 1
- The constant term to equal 1

So I have the system:
- $a_1 k^9 = \epsilon$
- $a_2 k^8 = 1$
- $a_3 = 1$

From equation (3):
$a_3 = 1$

From equation (2):
$k^8 = \frac{1}{a_2}$, so
$k = \left(\frac{1}{a_2}\right)^{1/8}$
$= \frac{1}{a_2^{1/8}}$

Substituting this into equation (1):
$\epsilon = a_1 k^9 = \frac{a_1}{a_2^{9/8}}$

FINAL ANSWER: a_1/a_2^(9/8)

Error: The same structural failure occurs with different polynomial degrees. The model again normalizes one coefficient independently while assuming $a_3 = 1$, so the resulting ϵ incorrectly contains no dependence on a_3.

G.1.2 Retrieved Critical Constraint

Both problems retrieve the same critical constraint capturing the coupled scaling relationships required for nondimensionalization.

Retrieved Memory: Coupled Scaling Constraint

```
Critical constraint:

The transformation must preserve the polynomial structure
    such that after nondimensionalization, the coefficient of
     the highest degree term becomes ϵ, the coefficient of the
     second highest degree term becomes 1, and the constant
    term becomes 1, with all original coefficients related
    through the transformation parameters.

Verification strategy:

Check that the transformation relations
```

$$a_1 = \epsilon \cdot L^m \cdot M,$$
$$a_2 = L^n \cdot M,$$
$$a_3 = M$$

```
are consistent with the target form ϵy^m + y^n + 1, where L and
    M are the scaling parameters for x and y, respectively.

Why this is critical:

Failure to satisfy this constraint means the
    nondimensionalization did not properly transform the
    original polynomial into the target form, either by
    incorrectly handling the scaling relationships between
    the coefficients or by not accounting for how the
    transformation affects all terms simultaneously. This
    would result in an incorrect expression for ϵ that does
    not arise from the proper scaling analysis of the
    original polynomial structure.
```

G.1.3 Constraint-Guided Inference

Using the retrieved constraint, the model correctly accounts for both scaling parameters and recovers the dependence of ϵ on all original coefficients.

Constraint Guided Inference - Question 1

Step 1 - Extract:

Relevant equations and conditions from the Critical Constraint Analysis:
- Original polynomial: $a_1x^m + a_2x^n + a_3$
- Target form: $\epsilon y^m + y^n + 1$
- Transformation relationships:

$$\begin{aligned} a_1 &= \epsilon \cdot L^m \cdot M, \\ a_2 &= L^n \cdot M, \\ a_3 &= M \end{aligned}$$

Step 2 - Solve:

We want to express ϵ in terms of a_1, a_2, a_3.

From the third equation:
$M = a_3$

Substitute $M = a_3$ into the second equation:
$a_2 = L^n \cdot a_3 \implies L^n = \frac{a_2}{a_3} \implies L = \left(\frac{a_2}{a_3}\right)^{1/n}$

Now substitute both $M = a_3$ and
$L = \left(\frac{a_2}{a_3}\right)^{1/n}$
into the first equation:

$$\begin{aligned} a_1 &= \epsilon \cdot \left(\left(\frac{a_2}{a_3}\right)^{1/n}\right)^m \cdot a_3 \\ &= \epsilon \cdot \left(\frac{a_2}{a_3}\right)^{m/n} \cdot a_3 \end{aligned}$$

Solve for ϵ:

$$\begin{aligned} \epsilon &= \frac{a_1}{a_3} \cdot \left(\frac{a_2}{a_3}\right)^{m/n} \\ &= \frac{a_1}{a_3} \cdot \left(\frac{a_3}{a_2}\right)^{m/n} \end{aligned}$$

For $m = 7$ and $n = 2$:
$\epsilon = \frac{a_1}{a_3}\left(\frac{a_3}{a_2}\right)^{7/2}$

Step 3 - Verify:

Plug back into the original transformation:

$$a_1 = \epsilon \cdot L^7 \cdot M$$
$$= \left[\frac{a_1}{a_3}\left(\frac{a_3}{a_2}\right)^{7/2}\right]\left[\left(\frac{a_2}{a_3}\right)^{1/2}\right]^7 a_3$$

Simplify:

$$\frac{a_1}{a_3}\left(\frac{a_3}{a_2}\right)^{7/2}\left(\frac{a_2}{a_3}\right)^{7/2} a_3$$
$$= \frac{a_1}{a_3}\left(\frac{a_3}{a_2}\cdot\frac{a_2}{a_3}\right)^{7/2} a_3$$
$$= \frac{a_1}{a_3}\cdot 1^{7/2}\cdot a_3 = a_1$$

So the transformation is consistent.

FINAL ANSWER: $\boxed{\frac{a_1}{a_3}\left(\frac{a_3}{a_2}\right)^{7/2}}$

Constraint Guided Inference - Question 2

Step 1 - Extract:

Relevant equations and conditions from the Critical Constraint Analysis:
- Original polynomial: $a_1x^m + a_2x^n + a_3$
- Target form: $\epsilon y^m + y^n + 1$
- Transformation relationships:

$$a_1 = \epsilon \cdot L^m \cdot M,$$
$$a_2 = L^n \cdot M,$$
$$a_3 = M$$

Step 2 - Solve:

We are given:

$$a_1 = \epsilon \cdot L^m \cdot M,$$
$$a_2 = L^n \cdot M,$$
$$a_3 = M$$

From the third equation:
$M = a_3$

Substitute $M = a_3$ into the second equation:

$a_2 = L^n \cdot a_3 \Rightarrow L^n = \frac{a_2}{a_3} \Rightarrow L = \left(\frac{a_2}{a_3}\right)^{1/n}$

Now substitute both expressions for L and M into the first equation:

$$\begin{aligned} a_1 &= \epsilon \cdot \left(\left(\frac{a_2}{a_3}\right)^{1/n}\right)^m \cdot a_3 \\ &= \epsilon \cdot \left(\frac{a_2}{a_3}\right)^{m/n} \cdot a_3 \end{aligned}$$

Solve for ϵ:

$$\begin{aligned} \epsilon &= \frac{a_1}{a_3} \cdot \left(\frac{a_2}{a_3}\right)^{m/n} \\ &= \frac{a_1}{a_3} \cdot \left(\frac{a_3}{a_2}\right)^{m/n} \end{aligned}$$

For $m = 9$ and $n = 8$:

$\epsilon = \frac{a_1}{a_3}\left(\frac{a_3}{a_2}\right)^{9/8}$

Step 3 - Verify:

Plug back into the original transformation:

$$\begin{aligned} a_1 &= \epsilon \cdot L^9 \cdot M \\ &= \left[\frac{a_1}{a_3}\left(\frac{a_3}{a_2}\right)^{9/8}\right]\left[\left(\frac{a_2}{a_3}\right)^{1/8}\right]^9 a_3 \end{aligned}$$

Simplify:

$$\begin{aligned} &\frac{a_1}{a_3}\left(\frac{a_3}{a_2}\right)^{9/8}\left(\frac{a_2}{a_3}\right)^{9/8} a_3 \\ &= \frac{a_1}{a_3}\left(\frac{a_3}{a_2} \cdot \frac{a_2}{a_3}\right)^{9/8} a_3 \\ &= \frac{a_1}{a_3} \cdot 1^{9/8} \cdot a_3 = a_1 \end{aligned}$$

So the transformation is consistent.

FINAL ANSWER: $\boxed{\frac{a_1}{a_3}\left(\frac{a_3}{a_2}\right)^{9/8}}$

G.2 Asymptotic Integration with Programmatic Constraints

G.2.1 Base Model Failures

The base model exhibits two distinct failures in identifying and handling the dominant contribution in Laplace asymptotics.

Question 1

Consider the integral

$$I(x) = \int_0^{0.6} \left(2.9t^5 + 1.8t^4 + 2.8\,\mathrm{atan}(t)\right) e^{-x(2.6t^5+1.9t^4-1.4t^2)}\, dt.$$

Develop an analytical formula for $I(x)$ that is accurate as $x \to \infty$.

Hint: Answer with a SymPy convertible formula in variable x.

Base Model Inference - Question 1

```
REASONING:

To develop an analytical formula for I(x) as x approaches
    infinity,
I need to apply asymptotic analysis techniques, specifically
    Laplace's
method or saddle point method, since the integrand contains
    an
exponential term with a large parameter x.

The key steps are:
1. Identify the dominant contribution to the integral by
    examining
   the exponent's behavior.
2. Find the critical points of the exponent's argument.
3. Apply Laplace's method or a similar asymptotic technique.
4. Express the leading-order asymptotic behavior in terms of
    x.

CODE:

import sympy as sp

def step():
    t, x = sp.symbols('t x')

    f_t = 2.9*t**5 + 1.8*t**4 + 2.8*sp.atan(t)
    g_t = 2.6*t**5 + 1.9*t**4 - 1.4*t**2

    dg_dt = sp.diff(g_t, t)
    critical_points = sp.solve(dg_dt, t)

    real_critical_points = [
```

```
        p for p in critical_points
        if p.is_real and 0 <= p <= 0.6
    ]

    if real_critical_points:
t0 = real_critical_points[0]

        d2g_dt2 = sp.diff(g_t, t, 2).subs(t, t0)

        numerator = f_t.subs(t, t0)
        denominator = sp.sqrt(
            x * abs(d2g_dt2.subs(t, t0))
        )

        result = (
            numerator / denominator
            * sp.exp(-x * g_t.subs(t, t0))
        )
    else:
        result = (
            sp.Symbol('C')
            * sp.exp(-x * g_t.subs(t, 0.6))
            / sp.sqrt(x)
        )

    return result

EXECUTION RESULT:
0
```

Error: The program selects the first admissible critical point instead of comparing all critical points and endpoints to identify the globally dominant contribution. It consequently selects $t_0 = 0$, where the prefactor vanishes, and returns zero.

Question 2

Consider the integral

$$I(x) = \int_{-0.8}^{0.9} \left(-1.6t^4 - 2.6t^2 - 0.9\cos(t) - 1.9\right) e^{x(-3.0t^5 + 0.4t^2 - 1.4t)}\, dt.$$

Develop an analytical formula for $I(x)$ that is accurate as $x \to \infty$.

Hint: Answer with a SymPy convertible formula in variable x.

Base Model Inference - Question 2

REASONING:

```
To develop an analytical formula for I(x) as x approaches
   infinity,
I need to apply Laplace's method for asymptotic evaluation of
integrals.

The key steps are:
1. Identify the dominant contribution by examining the
   exponential
   term.
2. Find the maximum of the exponent on the integration
   interval.
3. Apply Laplace's method at the maximizing point.
4. Compute the required derivatives.
5. Express the leading-order asymptotic form in terms of x.

CODE:

import sympy as sp

def step():
    t, x = sp.symbols('t x')

    f_t = -3.0*t**5 + 0.4*t**2 - 1.4*t

    f_prime = sp.diff(f_t, t)
    critical_points = sp.solve(f_prime, t)

    valid_critical_points = [
        p for p in critical_points
        if p.is_real and -0.8 <= p <= 0.9
    ]

    max_val = float('-inf')
    max_point = None

    for point in valid_critical_points:
        val = f_t.subs(t, point).evalf()
        if val > max_val:
            max_val = val
            max_point = point

    if max_point is None:
        boundary_values = []

        for bound in [-0.8, 0.9]:
            val = f_t.subs(t, bound).evalf()
            boundary_values.append((val, bound))

        max_val, max_point = max(
            boundary_values,
            key=lambda x: x[0]
        )

    f_double_prime = sp.diff(f_t, t, 2)
    second_deriv_at_max = (
        f_double_prime.subs(t, max_point)
    )
```

```
    g_t = (
        -1.6*t**4
        - 2.6*t**2
        - 0.9*sp.cos(t)
        - 1.9
    )

    g_at_max = g_t.subs(t, max_point)

result = g_at_max *
  sp.sqrt(2*sp.pi/sp.Abs(second_deriv_at_max))
* sp.exp(x * f_t.subs(t, max_point)) / sp.sqrt(x)

    return result
```

EXECUTION RESULT:
$-1.22078952570137\sqrt{\pi}\,e^{2.35904x}/\sqrt{x}$

Error: The program correctly identifies an endpoint as dominant but still applies the Gaussian interior-point Laplace formula. It therefore produces an $x^{-1/2}$ prefactor instead of the correct x^{-1} endpoint scaling.

G.2.2 Retrieved Critical Constraint

Both problems retrieve the same constraint governing global dominance and the appropriate local asymptotic form.

Retrieved Memory: Global Laplace Dominance Constraint

Critical constraint:

The asymptotic expansion of $I(x)$ as $x \to \infty$ must match the leading-order behavior obtained by applying Laplace's method to the dominant contribution of the integrand, where the exponential term $e^{x\phi(t)}$ is analyzed through its global maximum on the interval $[a, b]$.

Verification strategy:

For the given integral $I(x)$, identify the exponent function $\phi(t)$ and the non-exponential prefactor $f(t)$. Find the critical points of $\phi(t)$ by solving $\phi'(t) = 0$, and consider these together with the endpoints a and b to determine the point t_0 that globally maximizes $\phi(t)$ on the interval $[a, b]$.

If t_0 is an interior point, compute $\phi''(t_0)$ and verify

$$I(x) \sim f(t_0)e^{x\phi(t_0)}\left(\frac{2\pi}{x|\phi''(t_0)|}\right)^{1/2}.$$

If the maximum occurs at the right endpoint b with $\phi'(b) > 0$, verify

$$I(x) \sim \frac{f(b)e^{x\phi(b)}}{x\phi'(b)}.$$

If the maximum occurs at the left endpoint a with $\phi'(a) < 0$, verify

$$I(x) \sim \frac{f(a)e^{x\phi(a)}}{x|\phi'(a)|}.$$

If the candidate does not match the appropriate leading-order form, it must be rejected.

Why this is critical:

Failure to properly identify the global maximum of $\phi(t)$, including whether it occurs at an interior critical point or an endpoint, will result in an incorrect asymptotic scaling with x. The leading-order behavior must be determined by the global maximum of the exponent function $\phi(t)$ on the integration domain, not by a non-dominant critical point or an incorrect local balance. Any candidate that does not correctly capture this maximum and the corresponding local behavior fails to satisfy the fundamental requirement of Laplace's method for the given integral structure.

Verification mode: executable_verifier_code

G.2.3 Constraint-Guided Programmatic Inference

Using the retrieved constraint, the programmer explicitly compares all candidate dominant points and applies the asymptotic form corresponding to the selected regime.

Constraint Guided Inference - Question 1

```
import sympy as sp

def solver():
    t, x = sp.symbols('t x')

    f_t = (
        2.9*t**5
        + 1.8*t**4
        + 2.8*sp.atan(t)
```

```
    )

    phi_t = -(
        2.6*t**5
        + 1.9*t**4
        - 1.4*t**2
    )

    phi_prime = sp.diff(phi_t, t)
    critical_points = sp.solve(phi_prime, t)

    valid_critical_points = [
        pt for pt in critical_points
        if pt.is_real and 0 <= pt <= 0.6
    ]

endpoints = [0, 0.6]

candidates = valid_critical_points + endpoints

    phi_values = [
        (pt, phi_t.subs(t, pt).evalf())
        for pt in candidates
    ]

max_point = max(phi_values, key=lambda x:  x[1])

    t_max = max_point[0]

is_endpoint = t_max in endpoints

    if is_endpoint:
        if t_max == 0:
            phi_prime_at_0 = phi_prime.subs(t, 0)

            if phi_prime_at_0 < 0:
                result = (
                    f_t.subs(t, 0)
                    * sp.exp(
                        x * phi_t.subs(t, 0)
                    )
                    / (
                        x
                        * abs(phi_prime_at_0)
                    )
                )
            else:
                raise ValueError(
                    "Invalid endpoint condition"
                )

        elif t_max == 0.6:
            phi_prime_at_06 = (
                phi_prime.subs(t, 0.6)
            )

            if phi_prime_at_06 > 0:
```

```
                result = (
                    f_t.subs(t, 0.6)
                    * sp.exp(
                        x * phi_t.subs(t, 0.6)
                    )
                    / (
                        x
                        * phi_prime_at_06
                    )
                )
            else:
                raise ValueError(
                    "Invalid endpoint condition"
                )

    else:
        phi_double_prime = sp.diff(
            phi_t, t, 2
        )

        phi_double_prime_at_tmax = (
            phi_double_prime.subs(
                t, t_max
            )
        )

        f_at_tmax = f_t.subs(t, t_max)

        exp_term = sp.exp(
            x * phi_t.subs(t, t_max)
        )

 sqrt_term = sp.sqrt(2 * sp.pi /
 (x * abs(phi_double_prime_at_tmax)))

        result = (
            f_at_tmax
            * exp_term
            * sqrt_term
        )

    return f"final answer: {result}"

final_answer = solver()
```

FINAL ANSWER: $0.719832722044118\sqrt{\pi}\sqrt{\frac{1}{x}}e^{0.157699686244314x}$

Constraint Guided Inference - Question 2

```
import sympy as sp

def solver():
    t, x = sp.symbols('t x')
```

```
    f_t = (
        -1.6*t**4
        - 2.6*t**2
        - 0.9*sp.cos(t)
        - 1.9
    )

    phi_t = (
        -3.0*t**5
        + 0.4*t**2
        - 1.4*t
    )

    integrand = (
        f_t
        * sp.exp(x * phi_t)
    )

    phi_prime = sp.diff(phi_t, t)
    critical_points = sp.solve(
        phi_prime, t
    )

    a, b = -0.8, 0.9

endpoints = [a, b]

all_points = [p.evalf() for p in critical_points
if p.is_real and a <= p.evalf() <= b] + endpoints

    phi_values = [
        (
            point,
            phi_t.subs(t, point).evalf()
        )
        for point in all_points
    ]

max_point = max(phi_values, key=lambda x:  x[1])

    t_max = max_point[0]
    phi_max = max_point[1]

is_endpoint = t_max in [a, b]

    f_at_tmax = (
        f_t.subs(t, t_max).evalf()
    )

    if not is_endpoint:
        phi_double_prime = sp.diff(
            phi_t, t, 2
        )

        phi_double_prime_at_tmax = (
            phi_double_prime
            .subs(t, t_max)
```

```
            .evalf()
        )

        sqrt_term = sp.sqrt(
            2 * sp.pi
            / (
                x
                * abs(
                    phi_double_prime_at_tmax
                )
            )
        )

        result = (
            f_at_tmax
            * sp.exp(x * phi_max)
            * sqrt_term
        )

    else:
if t_max == b:

            phi_prime_at_b = (
                phi_prime
                .subs(t, b)
                .evalf()
            )

result = f_at_tmax * sp.exp(x * phi_max)
/ (x * abs(phi_prime_at_b))

        else:
            phi_prime_at_a = (
                phi_prime
                .subs(t, a)
                .evalf()
            )

            result = (
                f_at_tmax
                * sp.exp(x * phi_max)
                / (
                    x
                    * abs(phi_prime_at_a)
                )
            )

    result = sp.simplify(result)

    return f"final answer: {result}"

final_answer = solver()
```

FINAL ANSWER: $-\frac{0.592179379082655\, e^{2.35904x}}{x}$

H Programmatic Verification and Error Correction

H.1 Verifier-Guided Revision

This example illustrates how executable verification can prevent an incorrect intermediate solution from being accepted and provide feedback for subsequent revision.

Question

Nondimensionalize the polynomial

$$P(x) = -10x^8 - 4x^2 + 2$$

into a polynomial of the form

$\epsilon y^8 \pm y^2 \pm 1$.

Solve for ϵ.

Hint: Answer with a floating-point number with 2 decimal places.

H.1.1 Initial Program and Verification Failure

The initial program normalizes the (*@y^2@*) coefficient in isolation and derives a candidate value for (*@ϵ@*). Before returning the answer, however, it invokes the executable verifier.

Initial Constraint-Guided Program

```
def solver():
    # From the y^2 coefficient:
    # -4 = -alpha^2
    # alpha^2 = 4
    # alpha = 2

 epsilon = -10 / 256

    # Verify the candidate before returning it
 if verify_nondimensionalization(epsilon):
        return f"final answer: {epsilon:.2f}"
    else:
 raise ValueError("Verification failed")

final_answer = solver()
```

The candidate fails because the implied normalization does not satisfy the required coefficient relationships simultaneously.

Verifier Feedback

```
STATE: program_result
STATUS: failure
ERROR_TYPE: ValueError
```

```
ERROR: Verification failed
```

H.1.2 VERIFIER-GUIDED REVISION

The failed verification prevents the incorrect candidate from being returned. During revision, the model reconsiders the normalization and introduces an overall scaling factor so that the constant and quadratic terms are normalized jointly.

Revised Program

```
def solver():
    # Introduce an overall scaling factor C:
    #
    # -10*alpha^8 = epsilon*C
    # -4*alpha^2  = +/- C
    #  2          = +/- C

 # From the constant term, choose C = -2
 # Then -4*alpha^2 = -2, so alpha^2 = 1/2

 alpha_val = sp.sqrt(2)/2

    # -10*alpha^8 = epsilon*C with C = -2
    # epsilon = 5*alpha^8

 epsilon_val = 5 * (alpha_val**8)

    epsilon_decimal = float(epsilon_val)

    return round(epsilon_decimal, 2)

final_answer = solver()

FINAL ANSWER: 0.31
```

I EXAMPLE TRAJECTORIES FOR ERROR ANALYSIS

This section reports two unsuccessful HARDMath2 trajectories from the complete CROW evaluation. We reproduce the orchestration decisions, information revealed by the environment, solver route, submitted answer, and evaluation outcome. The first trajectory shows an overly permissive memory match. The second shows the complementary coverage failure: the retrieved candidate is rejected, but no replacement constraint is constructed.

I.1 ACCEPTED MEMORY WITH NON-TRANSFERABLE ASSUMPTIONS

Problem

Find the leading-order WKB approximation for

$$\epsilon^2 y''(x) - (1 + \cos x) y(x) = 0$$

with $y(0) = 0$ and

$y'(0) = 2^{5/4}\cosh(1)/\epsilon$, where
$0 < \epsilon \ll 1$.

Retrieved Candidate (Environment)

MEMORY ID: M0005
RETRIEVAL SCORE: 0.6515

RELATED QUESTION:
Find a uniformly valid approximation to

$$\epsilon y''(x) + a(x)y'(x) + b(x)y(x) = 0$$

with prescribed boundary conditions as $\epsilon \to 0^+$.

The associated memory concerns boundary-layer location. It requires the outer
solution to use the condition away from the boundary layer and the remaining
condition to be enforced by a rapidly varying correction that decays into the
interior.

Step 1: Memory Judgment (Orchestrator)

ACTION: JUSTIFY_MEMORY

STRUCTURAL_COMPARISON:
The original and related problems are second-order differential equations
with a small parameter in the highest derivative and endpoint conditions.
The orchestrator states that both ask for a leading-order WKB approximation.

VERDICT: MATCH

ENVIRONMENT RESPONSE:
STATE: memory_justified. The next action must be READ_MEMORY.

Step 2: Memory Activation (Orchestrator and Environment)

ACTION: READ_MEMORY

ORCHESTRATOR JUSTIFICATION:
The related question is structurally similar because it is a second-order
equation with a small parameter in the highest derivative and requires a
leading-order WKB approximation.

ENVIRONMENT RESPONSE:

```
STATE: memory_ready
MEMORY_ID: M0005
ACTIVE_VERIFICATION_MODE: natural_language_reasoning_text

ACTIVE CONSTRAINT:
First derive the reduced outer solution. Determine which
   endpoint contains
the boundary layer before applying a boundary condition. Use
   the condition at
the opposite endpoint to fix the outer solution, and enforce
   the remaining
condition with a correction localized near the boundary-layer
    endpoint and
decaying into the interior.

The next action must be CALL_NATURAL_LANGUAGE with
   use_constraint: true.
```

Step 3: Constraint-Guided Solver Call

```
ACTION: CALL_NATURAL_LANGUAGE
use_constraint: true

PROMPT:
Derive the outer solution, determine which endpoint contains
   the boundary
layer, use the opposite boundary condition to fix the outer
   solution, and
construct a localized, decaying boundary-layer correction.

ENVIRONMENT RESPONSE:
STATUS: malformed
The Natural Language Operator did not return a FINAL ANSWER
   marker. The
previous response was retained for revision.
```

Steps 4–7: Formatting Revisions

```
ACTION: CALL_NATURAL_LANGUAGE_REVISER
use_constraint: true

The orchestrator requested a shorter response containing a
   FINAL ANSWER and a
boxed expression. Four consecutive attempts returned
STATUS: malformed_revision because the required marker was
   still absent.
```

Step 8: Successful Revision

```
ACTION: CALL_NATURAL_LANGUAGE_REVISER
use_constraint: true
```

RESULT:

$$\boxed{y(x) = -2^{3/4}\cosh(1)\exp\left[-\frac{2\sqrt{2}}{\epsilon}\sin\left(\frac{x}{2}\right)\right]}$$

Step 9: Submission and Evaluation

ACTION: SAVE_ANSWER

REFERENCE ANSWER:

$$y(x) \sim 2\cosh(1)(1+\cos x)^{-1/4}\sinh\left(\frac{2\sqrt{2}\sin(x/2)}{\epsilon}\right).$$

EVALUATOR VERDICT: INCORRECT

The submitted expression omits the WKB amplitude $(1+\cos x)^{-1/4}$ and retains only one exponential branch, so it cannot satisfy both initial conditions. The accepted memory instead describes a reduced first-order outer problem and an endpoint boundary-layer correction. The match was therefore based on broad surface features rather than the applicable asymptotic structure.

I.2 Rejected Memory, No Replacement Constraint, and Direct Failure

Problem

Find the leading-order WKB approximation for

$$\epsilon^2 y'' = (x+1)y$$

with $y(0) = 0$ and $y'(0) = 1$, where $0 < \epsilon \ll 1$.

Retrieved Candidate (Environment)

MEMORY ID: M0005
RETRIEVAL SCORE: 0.6829

The top retrieval is again the boundary-layer memory for

$$\epsilon y''(x) + a(x)y'(x) + b(x)y(x) = 0.$$

No WKB-specific memory is retrieved.

Step 1: Memory Judgment (Orchestrator)

```
ACTION: JUSTIFY_MEMORY

STRUCTURAL_COMPARISON:
The original problem has no first-order term and supplies
    initial conditions,
whereas the related problem includes a first-order term and
    asks for a
uniformly valid boundary-layer approximation. The structures
    are similar but
not identical.

VERDICT: NO_MATCH

ENVIRONMENT RESPONSE:
STATE: memory_justified. Memory does not match. The next
    action must be
JUSTIFY_CONSTRAINT_NEED.
```

Step 2: Constraint Decision (Orchestrator)

```
ACTION: JUSTIFY_CONSTRAINT_NEED

CONSTRAINT_NEED:
The problem asks for a leading-order WKB approximation with
    initial
conditions, rather than a uniformly valid boundary-layer
    approximation.
Therefore, no new constraint is required and the problem can
    be solved
directly.

VERDICT: NO_CONSTRAINT

ENVIRONMENT RESPONSE:
STATE: constraint_need_justified. The next action must call a
     solver with
use_constraint: false.
```

Step 3: Direct Solver Call

```
ACTION: CALL_NATURAL_LANGUAGE
use_constraint: false

PROMPT:
Find the leading-order WKB approximation for
ϵ²y″ = (x + 1)y with
y(0) = 0 and y′(0) = 1.

RESULT:
```

$$\boxed{y(x) \sim \frac{1}{\sqrt{x+1}} \exp\left[\frac{2}{3\epsilon}\left((x+1)^{3/2}-1\right)\right]}$$

Step 4: Submission and Evaluation

```
ACTION: SAVE_ANSWER

REFERENCE ANSWER:
```

$$y(x) \sim \epsilon(x+1)^{-1/4} \sinh\left[\frac{2\left((x+1)^{3/2}-1\right)}{3\epsilon}\right].$$

```
EVALUATOR VERDICT: INCORRECT

The direct solver uses the wrong WKB amplitude and keeps only
    the growing
exponential branch. The missing decaying branch and
   normalization prevent it
from satisfying the initial conditions. Here the orchestrator
    correctly
rejects the unrelated boundary-layer memory, but the
   available memory provides
no WKB-specific guidance and the policy declines to construct
    a replacement
constraint.
```

These cases expose complementary limitations. Retrieval and policy selection must consider the assumptions under which a stored constraint transfers, not only coarse task-family similarity. Conversely, selective rejection is not sufficient when the memory bank lacks the needed reasoning pattern and direct reasoning remains unreliable.

J End-to-End Workflow Examples

We present two complete inference trajectories to illustrate how CROW coordinates memory selection, constraint-guided reasoning, and revision. The first example uses natural-language reasoning, while the second uses programmatic reasoning. In both cases, the orchestrator first verifies the structural relevance of the retrieved memory, activates its associated constraint, routes the problem to the appropriate solver, and invokes a specialized reviser when the initial solver attempt fails.

J.1 Natural-Language Workflow

This example illustrates a complete workflow using natural-language reasoning. The retrieved memory is first judged to be structurally compatible with the current problem. Its constraint is then activated and used by the natural-language solver. Although the initial reasoning derives the correct relationship, the response is malformed because it does not produce the required final-answer marker. The natural-language reviser repairs the response while preserving the constraint-guided derivation.

Problem

Nondimensionalize the polynomial

$$a_1 x^{10} + a_2 x^6 + a_3$$

into one of the form

$$\epsilon y^{10} + y^6 + 1.$$

Express ϵ as a function of a_1, a_2, and a_3.

Hint: Answer with a SymPy convertible formula with variables.

Step 1: Memory Justification (Orchestrator)

ACTION: JUSTIFY_MEMORY

STRUCTURAL_COMPARISON:

Both problems involve nondimensionalizing a polynomial with terms of
different powers of x into a form with terms of different powers of y,
with the goal of expressing epsilon as a function of the coefficients.

The original problem specifies exponents 10, 6, and 0, while the
retrieved problem uses the general exponents m, n, and 0.

VERDICT: MATCH

Step 2: Memory Activation (Orchestrator)

ACTION: READ_MEMORY

MEMORY ID: M0000

RELATED QUESTION:

Nondimensionalize the polynomial

$a_1 x^m + a_2 x^n + a_3$

into one of the form

$\epsilon y^m + y^n + 1$,

and express ϵ as a function of the original coefficients.

ACTIVE VERIFICATION MODE:

natural_language_reasoning_text

ACTIVE CONSTRAINT:

The transformation must preserve the complete polynomial structure so
that the highest-degree coefficient becomes epsilon, the second
coefficient becomes 1, and the constant term becomes 1.

The corresponding scaling relations are

$$a_1 = \epsilon L^m M, \qquad a_2 = L^n M, \qquad a_3 = M.$$

Step 3: Constraint-Guided Inference (Natural-Language Solver)

ACTION: CALL_NATURAL_LANGUAGE
use_constraint: true

Step 1 - Extract:

The active constraint gives

$$a_1 = \epsilon L^{10} M, \qquad a_2 = L^6 M, \qquad a_3 = M.$$

Step 2 - Solve:

From

$a_3 = M$

we obtain

$M = a_3$.

Using

$a_2 = L^6 M$

gives

$$L^6 = \frac{a_2}{a_3}, \qquad L = \left(\frac{a_2}{a_3}\right)^{1/6}.$$

Then

$$\epsilon = \frac{a_1}{L^{10}M} = \frac{a_1}{}\left(\frac{a_2}{a_3}\right)^{5/3} a_3.$$

The model continues simplifying the same expression but does not
produce the required final-answer marker.

STATUS: malformed

ERROR: Natural Language Operator did not return a FINAL ANSWER marker.

Step 4: Response Revision (Natural-Language Reviser)

ACTION: CALL_NATURAL_LANGUAGE_REVISER
use_constraint: true

REVISION INSTRUCTION:

The response is too long and does not include a final answer.
Please revise to provide a concise answer with the final answer marked.

REVISED RESPONSE:

From the active constraint,

$$a_3 = M, \qquad a_2 = L^6 M.$$

Therefore,

$$M = a_3, \qquad L^6 = \frac{a_2}{a_3}.$$

Hence,

$$L^{10} = \left(\frac{a_2}{a_3}\right)^{5/3}.$$

Substituting into

$a_1 = \epsilon L^{10} M$

gives

$$\epsilon = \frac{a_1}{}\left(\frac{a_2}{a_3}\right)^{5/3} a_3 = \frac{a_1}{a_2^{5/3}} a_3^{2/3}.$$

FINAL ANSWER: $\boxed{\frac{a_1}{a_2^{5/3}} a_3^{2/3}}$

Step 5: Answer Submission (Orchestrator)

ACTION: SAVE_ANSWER

Submitted answer: $\boxed{\frac{a_1}{a_2^{5/3}} a_3^{2/3}}$

J.2 Programmatic Workflow

This example illustrates the corresponding workflow when the retrieved memory specifies programmatic verification. After establishing that the retrieved Laplace-method memory matches the problem structure, the orchestrator activates the memory and routes the problem to the Programmer. The initial program implements the correct global-dominance logic, but fails during execution because a floating-point endpoint is incorrectly treated as a symbolic object. The Programmer Reviser repairs this local execution error while preserving the constraint-guided asymptotic reasoning.

Problem

Consider the integral

$$I(x) = \int_{0.1}^{0.8} \left(-2.4t^2 - 3.6t - 2.1\,\mathrm{atan}(t)\right) e^{-x(-0.9t^4+0.5t^3+2.8t^2)}\, dt.$$

Develop an analytical formula for $I(x)$ that is accurate as $x \to \infty$.

Hint: Answer with a SymPy convertible formula in variable x.

Step 1: Memory Justification (Orchestrator)

ACTION: JUSTIFY_MEMORY

STRUCTURAL_COMPARISON:

Both problems involve integrals of the form

$$\int_a^b f(t)e^{x\phi(t)}\, dt$$

with the goal of developing an analytical formula accurate as $x \to \infty$.

The original problem specifies concrete functions for $f(t)$ and $\phi(t)$, while the retrieved problem represents the same asymptotic structure in general form.

VERDICT: MATCH

Step 2: Memory Activation (Orchestrator)

ACTION: READ_MEMORY

MEMORY ID: M0004

RELATED QUESTION:

Consider an integral of the form

$$I(x) = \int_a^b f(t)e^{x\phi(t)}\,dt$$

and develop an analytical formula accurate as $x \to \infty$.

ACTIVE VERIFICATION MODE:

executable_verifier_code

ACTIVE CONSTRAINT:

The leading-order asymptotic behavior must be determined by the global maximum of $\phi(t)$ over the entire integration interval.

Critical points satisfying $\phi'(t) = 0$ must therefore be considered together with both endpoints.

If the maximum is interior, use

$$I(x) \sim f(t_0)e^{x\phi(t_0)}\left(\frac{2\pi}{x|\phi''(t_0)|}\right)^{1/2}.$$

If the maximum occurs at an endpoint, use the corresponding x^{-1} endpoint expansion.

Step 3: Constraint-Guided Inference (Programmer)

ACTION: CALL_PROGRAMMER
use_constraint: true

```
import sympy as sp
```

```
def solver():
    t, x = sp.symbols('t x')

    phi_t = (
        0.9*t**4
        - 0.5*t**3
        - 2.8*t**2
    )

    phi_prime = sp.diff(phi_t, t)
    phi_double_prime = sp.diff(phi_prime, t)

    critical_points = sp.solve(phi_prime, t)

endpoints = [0.1, 0.8]
candidates = critical_points + endpoints

valid_candidates = [pt for pt in candidates
if pt.is_real and 0.1 <= pt <= 0.8]

    valid_candidates.extend(endpoints)

    phi_values = [
        (pt, phi_t.subs(t, pt).evalf())
        for pt in valid_candidates
    ]

max_point = max(phi_values, key=lambda x:  x[1])

    t_max = max_point[0]
    phi_max = max_point[1]

    if t_max == 0.1:
        phi_prime_at_01 = (
            phi_prime.subs(t, 0.1).evalf()
        )

        if phi_prime_at_01 < 0:
            f_a = (
                -2.4 * 0.1**2
                - 3.6 * 0.1
                - 2.1 * sp.atan(0.1)
            ).evalf()

            result = (
                f_a
                * sp.exp(
                    x * phi_t.subs(t, 0.1)
                )
                / (
                    x
                    * abs(phi_prime_at_01)
                )
            )

    elif t_max == 0.8:
        phi_prime_at_08 = (
```

```
            phi_prime.subs(t, 0.8).evalf()
        )

        if phi_prime_at_08 > 0:
            f_b = (
                -2.4 * 0.8**2
                - 3.6 * 0.8
                - 2.1 * sp.atan(0.8)
            ).evalf()

            result = (
                f_b
                * sp.exp(
                    x * phi_t.subs(t, 0.8)
                )
                / (
                    x
                    * phi_prime_at_08
                )
            )

    else:
        f_t0 = (
            -2.4 * t_max**2
            - 3.6 * t_max
            - 2.1 * sp.atan(t_max)
        ).evalf()

        phi_double_prime_at_t0 = (
            phi_double_prime
            .subs(t, t_max)
            .evalf()
        )

        sqrt_term = sp.sqrt(
            2 * sp.pi
            / (
                x
                * abs(
                    phi_double_prime_at_t0
                )
            )
        )

        result = (
            f_t0
            * sp.exp(x * phi_max)
            * sqrt_term
        )

    return f"final answer: {result}"

final_answer = solver()

STATUS: execution failure

ERROR: 'float' object has no attribute 'is_real'
```

Step 4: Program Revision (Programmer Reviser)

```
ACTION: CALL_PROGRAMMER_REVISER
use_constraint: true

REVISION INSTRUCTION:

The error "'float' object has no attribute 'is_real'"
   indicates that the
candidate filtering logic applies a symbolic property to a
   floating-point
value. Repair this local failure while preserving the
   constraint-guided
solution.

REVISED CODE:

critical_points = sp.solve(phi_prime, t)

valid_candidates = []

for pt in critical_points:
    try:
 if isinstance(pt, float):
        if 0.1 <= pt <= 0.8:
            valid_candidates.append(pt)
    else:
 if pt.is_real and 0.1 <= pt <= 0.8:
            valid_candidates.append(pt)
    except AttributeError:
        continue

valid_candidates.extend([0.1, 0.8])

phi_values = [
    (pt, phi_t.subs(t, pt).evalf())
    for pt in valid_candidates
]

max_point = max(phi_values, key=lambda x:  x[1])

t_max = max_point[0]
phi_max = max_point[1]

if t_max == 0.1:
    phi_prime_at_01 = (
        phi_prime.subs(t, 0.1).evalf()
    )

    if phi_prime_at_01 < 0:
        f_a = (
            -2.4 * 0.1**2
            - 3.6 * 0.1
            - 2.1 * sp.atan(0.1)
        ).evalf()

 result = f_a * sp.exp(x * phi_t.subs(t, 0.1))
```

```
 / (x * abs(phi_prime_at_01))

EXECUTION STATUS: success

FINAL ANSWER: $-\frac{1.03833421461575\, e^{-0.02841x}}{x}$
```

Step 5: Answer Submission (Orchestrator)

```
ACTION: SAVE_ANSWER

Submitted answer: $-\frac{1.03833421461575\, e^{-0.02841x}}{x}$
```