LEARNING PROMPT CHAINS FOR FROZEN LLMS: INTER-CALL ORCHESTRATION BEYOND SINGLE-CALL REASONING

Anonymous authors
Paper under double-blind review

ABSTRACT

Reinforcement learning has made Chain-of-Thought prompting an effective form of test-time computation, but it remains unclear whether multi-call prompting strategies can be learned in the same way. We study prompt chaining as a trainable orchestration policy for frozen large language models (LLMs). We introduce Chainer, a lightweight model that maps each user request to an open-loop sequence of self-contained sub-prompts executed by a frozen LLM. Chainer is first initialized by supervised fine-tuning and then optimized with GRPO using end-task rewards, chain-quality signals, validity bonuses, and a dynamic step cost. Across six benchmarks, learned chaining improves a non-reasoning executor on four tasks. With reasoning enabled, it remains complementary on five, improving Arena-Hard (+17.3), AIME (+10.0), LiveCodeBench (+8.2), GPQA (+2.0), and AA-Omniscience (-2.9; lower is better), while degrading IFBench. Experiments shows that these gains are not redundant with longer single-call reasoning: gains persist when token budgets are matched. Since executor weights are fixed, the improvements arise solely from optimizing the prompt trajectory. These results position learned prompt chaining as a distinct axis for scaling frozen LLM systems.

1 INTRODUCTION

Recent progress in large language models (LLMs) has highlighted the importance of test-time computation: allocating additional inference-time compute can improve answer quality without changing model weights. The dominant form of this idea is intra-call reasoning, where a model spends more tokens within a single generation before producing an answer. Chain-of-Thought prompting (Wei et al., 2023) is an early example of this paradigm, and recent reinforcement-learning approaches have further internalized deliberation and verification into reasoning-style models (OpenAI et al., 2024; DeepSeek-AI et al., 2025; Abdin et al., 2025). As a result, test-time compute is often treated as largely synonymous with longer single-call reasoning.

Many practical LLM workflows, however, improve outputs through a different mechanism: decomposing a request into a sequence of prompts executed over multiple calls. This technique, commonly known as prompt chaining, is widely used in practice (Anthropic, 2024) and has been studied empirically in domains such as text summarization (Sun et al., 2024). However, designing effective chains requires expertise, the optimal decomposition varies across tasks and models, and long chains increase latency and cost. This motivates a learned Chainer that can generate efficient high-quality chains automatically.

In this context, we train a lightweight Chainer model that acts as an external orchestration layer for a frozen LLM. Given a user request, the Chainer rewrites it into an open-loop, self-contained sequence of sub-prompts, which are executed sequentially by the frozen executor model. We also enforce context isolation between the original user request and the executor, allowing the Chainer total strategic control to decide how to stage and reveal task information through the generated chain.

The Chainer is trained with an SFT cold start and subsequently optimized using GRPO (Shao et al., 2024). Its reward combines end-task outcome rewards, direct chain-quality rewards, JSON validity bonuses, and a dynamic step cost that penalizes unnecessary decomposition. This objective

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.

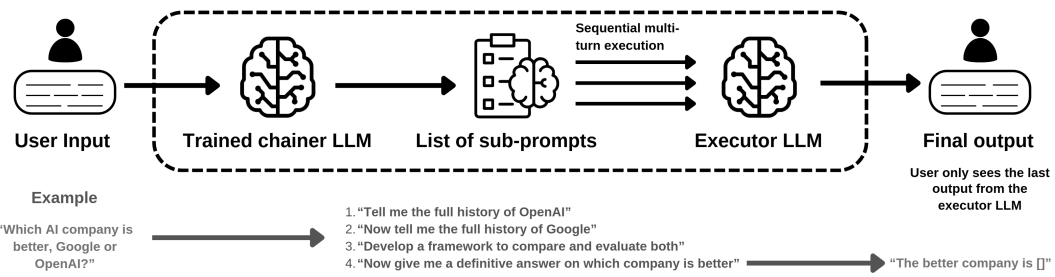


Figure 1: Overview of our automated prompt-chaining pipeline

encourages prompt trajectories that improve over direct execution while discouraging verbose chains when a shorter trajectory is sufficient.

Because the executor is kept frozen while only the Chainer is trained, our setup separates the policy that structures computation from the model that performs generation. This makes prompt chaining a distinct form of test-time compute from ordinary single-call reasoning: reasoning allocates more computation within one generation, whereas chaining allocates computation across multiple staged executor calls. The two mechanisms can also be combined, since the same generated chain can be executed either with reasoning disabled or with intrinsic reasoning enabled in the executor.

Accordingly, our empirical evaluation compares four regimes: direct execution without reasoning, direct execution with reasoning, chained execution without reasoning, and chained execution with reasoning. We also compare against strong hand-designed test-time baselines, including Least-to-Most prompting, Self-Refine, Tree-of-Thoughts (ToT), Chain-of-Thought (CoT) sampling, and Recursive Self-Aggregation (RSA). Across six benchmarks, learned chaining improves the non-reasoning executor on four tasks. When intrinsic reasoning is enabled, chaining further improves five tasks, while exposing failure modes on instruction following and hallucination-sensitive evaluation.

Our contributions are summarized as follows:

- **Learned test-time prompt chaining:** We introduce an RL-trained Chainer model that generates open-loop prompt trajectories for a frozen executor. The Chainer decomposes each user request into sub-prompts and learns when this decomposition improves end-task performance.
- **Synergy with Reasoning Models:** Across six benchmarks, learned chaining improves the non-reasoning executor on four tasks and the reasoning-enabled executor on five. This shows that inter-call orchestration can provide gains beyond longer single-call deliberation, while also revealing cases where decomposition can hurt.
- **Scaling and transfer analysis:** We show that the gains are not explained solely by increasing reasoning-token budgets, and that Chainer policies transfer imperfectly across executors. This suggests that learned prompt trajectories are partly adapted to the executor used during training.

2 Related work

Prompt decomposition and test-time compute. Prompt decomposition methods show that LLMs can benefit from solving a request through intermediate stages. Chain-of-Thought and reasoning-oriented RL allocate this computation within a single generation (Wei et al., 2023; OpenAI et al., 2024; DeepSeek-AI et al., 2025; Abdin et al., 2025), while methods such as Plan-and-Solve and Least-to-Most prompting use hand-written decompositions to expose simpler subproblems (Wang et al., 2023; Zhou et al., 2023a). Prompt chaining is also widely used in practice (Anthropic, 2024) and has been studied in domains such as summarization, information extraction, and document classification (Sun et al., 2024; Kwak et al., 2024; Trautmann, 2023). These works establish that staged prompting can help, but the staging procedure is usually manual, fixed at the task level, or expressed as a single-call reasoning style. Our work asks a different question: can the prompt

trajectory itself be learned as a test-time compute policy for a frozen executor, and does it still help when the executor already has intrinsic reasoning enabled?

Structured inference over frozen models. Several methods improve frozen LLMs by imposing additional inference-time structure. Self-Refine applies a fixed generate–critique–revise loop (Madaan et al., 2023), ToT searches over intermediate reasoning states (Yao et al., 2023a), and RSA uses candidate generation followed by aggregation and refinement (Venkatraman et al., 2025). Deep Language Networks show that stacked LLM calls with learned prompts can outperform shallow prompting (Sordoni et al., 2023). These methods motivate multi-call computation, but they either prescribe the inference algorithm or learn prompts tied to a fixed architecture. Chainer instead learns an input-conditioned prompt-chain policy: each request receives a fresh, variable-length sequence of executor-facing prompts, and behaviors such as decomposition, verification, revision, aggregation, or direct execution can emerge as instance-specific choices rather than as separately hand-designed procedures.

Prompt and workflow optimization. Automatic prompt-optimization methods such as APE, ProTeGi, OPRO, PromptBreeder, GEPA, and DSPy use feedback to improve prompts, demonstrations, or modular LM programs (Zhou et al., 2023b; Pryzant et al., 2023; Yang et al., 2024; Fernando et al., 2023; Agrawal et al., 2025; Khattab et al., 2023). These approaches primarily optimize reusable artifacts: a task-level instruction, a demonstration set, or a program structure that is applied across inputs. More input-adaptive methods, including Directional Stimulus Prompting and Instance-adaptive Zero-shot CoT, generate hints or select a prompt style for each instance (Li et al., 2023; Yuan et al., 2024). A systematic survey catalogues 57 automatic prompt-optimization methods; our audit finds that 53 require task-specific data, demonstrations, metrics, human feedback, prompt search, program construction, or policy training before evaluation on a new task (Ramnath et al., 2025). BPO (Cheng et al., 2024) and FIPO (Lu et al., 2025) are the two directly matched train-once per-input prompt transformations in our comparison, and both are evaluated. SPRIG (Zhang et al., 2024) is also evaluated as a reusable but static cross-task system prompt; GATE uses a different human-in-the-loop protocol. SFT and GRPO are established optimization techniques; our contribution lies in the object being learned: an input-conditioned, variable-length prompt trajectory. The SFT initialization underperforms direct execution on five of six benchmarks and gains emerge after outcome-based GRPO (Appendix A.6), showing that valid decomposition alone is insufficient. Chainer leaves the executor weights unchanged and reuses the learned policy across heterogeneous requests without benchmark-specific demonstrations, validation feedback, or parameter updates at deployment.

Open-loop orchestration versus agents. Task decomposition is also used in agentic systems such as ReAct, which interleave reasoning, actions, observations, and replanning in an external environment (Yao et al., 2023b). Chainer is related to these planning and agentic workflows, but studies a restricted setting: an upfront sequence of text prompts executed by one frozen executor, without tools or online replanning. We compare with Meta-Prompting (Suzgun & Kalai, 2024) and Plan-and-Solve+ as relevant text-mediated workflow baselines. LatentMAS (Zou et al., 2025) is also related, but communicates through hidden states or key–value (KV) cache access and therefore uses a different model-access interface. Chainer amortizes orchestration into a single upfront policy call that emits the executor-facing prompt trajectory. Separating orchestration from execution allows Chainer to be trained while the executor remains unchanged, isolating the surrounding inference procedure within each executor setting.

3 Prompt Chaining as Test-Time Orchestration

We formulate prompt chaining as a trainable form of test-time orchestration for frozen LLMs. Instead of asking a model to solve a request in a single generation, our framework separates the policy that structures computation from the model that performs generation. A lightweight Chainer learns to produce an open-loop, self-contained sequence of prompts, and a frozen executor runs these prompts sequentially. This lets us study inter-call orchestration as a distinct axis of test-time compute from intra-call reasoning.

3.1 Problem Setup: Intra-Call vs. Inter-Call Compute

We study two ways to spend test-time compute with a frozen executor. Let x be the original user request, and let E_m denote the executor, where $m \in \{0, 1\}$ indicates whether intrinsic reasoning is disabled or enabled. In direct execution, the answer is produced in a single call,

$$y = E_m(x).$$

When reasoning is enabled, this increases computation within one generation, but the model still starts from the same initial context.

Our method instead allocates compute across multiple calls. A Chainer policy π_θ maps the request x to an open-loop prompt trajectory

$$\gamma = (a_1, \ldots, a_T) \sim \pi_\theta(\cdot \mid x),$$

where each a_t is a self-contained sub-prompt. Let o_t be the executor output at step t, and let h_t denote the accumulated execution history. Execution evolves as

$$o_t = E_m(h_{t-1} \oplus a_t), \qquad h_t = h_{t-1} \oplus a_t \oplus o_t,$$

with $h_0 = \varnothing$, and the final answer returned to the user is $y = o_T$.

This formulation captures the central distinction of the paper. Intra-call reasoning improves how much computation the executor performs within a single context, whereas chaining changes the sequence of contexts from which later calls are made. Intermediate steps therefore need not solve the task directly; their role can be to construct a more useful history for later execution, so that the final answer becomes easier for the frozen executor to generate. We refer to the induced history trajectory $(h_0, \ldots, h_T)$ as a semantic walk.

3.2 Chainer–Executor Architecture

Our system consists of two models operating in a strict open-loop configuration. The Chainer is a parameter-efficient Low-Rank Adaptation (LoRA) fine-tuned model based on Qwen-3 4b. Given the original user request, it outputs a structured JSON object:

$$\{\text{"}\texttt{walk}\text{"} : [\texttt{a}_1, \texttt{a}_2, \ldots, \texttt{a}_\texttt{T}]\}.$$

The **Executor** is a frozen Qwen-3 4b model that executes the generated prompts sequentially. The Chainer never directly answers the user; only the executor's final output is returned.

This architecture isolates task structuring from text generation. The Chainer decides how to decompose, order, and phrase intermediate prompts, while the frozen executor performs generation at each step. The same generated chain can be executed with reasoning disabled or enabled in the executor, allowing us to compare inter-call orchestration with intra-call reasoning directly. This separation confines training to the lightweight Chainer while allowing a potentially much larger or externally hosted executor to remain unchanged.

3.3 Chainer Constraints

To make the Chainer behave as a reliable orchestration policy rather than as a conversational assistant, we enforce several execution constraints. The Chainer must output only the JSON walk and is forbidden from answering the user directly. Each step must be a prompt written as a direct first-person request, so the executor can act on it as if talking to the user. We also enforce a blind-executor constraint: the executor never sees the original user prompt directly, so all relevant context and constraints must be revealed through the generated walk. Appendix A.7 shows that relaxing this constraint at inference time has no clear overall effect, but it remains useful during RL training to expand the space of decompositions the Chainer can explore.

We further restrict the chain to prevent common failure modes. The Chainer may not leak the solution inside its output, ask the user for clarification, or terminate on an intermediate artifact. The final step must request the complete user-facing deliverable. To control inference cost, redundant decomposition is discouraged and trivial requests may be handled with a single step. For large immutable inputs such as code, logs, or datasets, we use symbolic blob placeholders inside the walk, which are deterministically expanded by the runtime system before executor calls.

For multi-turn use, Chainer receives the prior user-visible conversation and the current request, then generates a fresh chain for that turn. After execution, intermediate prompts and outputs are discarded; prior user messages and final assistant answers are retained. Our experiments evaluate individual requests and do not establish robustness under extended multi-turn interaction, tool use, or long-context processing.

3.4 Learning the Chainer Policy

We train the Chainer in two stages. First, we perform supervised fine-tuning as a cold start to instill the structural constraints of the Chainer, which the base model consistently failed to satisfy.

We curate a synthetic SFT dataset using GPT-5.2 as a teacher model. For each prompt in the multi-domain base dataset, the teacher generates 6 to 10 distinct valid walks. These walks vary in length and strategy, ranging from single-step direct execution to longer multi-step decompositions. This exposes the model to multiple valid ways to decompose the same problem and provides a diverse exploration space for RL. We also add a set of manually written trajectories to correct observed edge cases and improve variety.

The resulting SFT policy reliably produced valid, executable JSON chains that respected the blind-executor constraint, providing a stable starting policy for RL.

Second, we optimize the Chainer with Group Relative Policy Optimization (GRPO). We train jointly across five domains: Creative Writing, Math, Code, General Conversation, and Multiple Choice Science questions. The training data is ordered by difficulty to provide a simple curriculum, and we enforce a maximum walk length of 8 steps throughout training.

During RL, the Chainer samples 10 distinct walks to explore the action space, while the executor is kept deterministic with temperature 0. This makes the trajectory induced by a given walk stable across repeated execution. For open-ended domains, we use a rubric-based LLM judge to evaluate the final output relative to the one-shot baseline, with a deterministic two-pass A/B swapping protocol to reduce position bias. Within the GRPO loop, we use `qwen/qwen3-30b-a3b-instruct-2507` for reward evaluation as it offers a better latency-cost trade-off for repeated reward calls.

The reward combines end-task improvement over a one-shot baseline, direct chain rewards, a JSON validity bonus, and a dynamic step cost:

$$R = \underbrace{(\text{Score}(y_{\text{walk}}) - \text{Score}(\bar{y}))}_{\text{Relative Quality Reward}} + \underbrace{\text{Score}_{\text{chain}}(\gamma)}_{\text{Direct Chain Reward}} + \underbrace{R_{\text{bonus}}}_{\text{JSON Validity Bonus}} - \underbrace{\lambda \cdot (T-1)}_{\text{Step Cost}}.$$

where:

- **Relative Quality Reward:** Domain-dependent, with final values in $[-1, 1]$.
  - **Verifiable Domains (Math, Code):** Binary scoring (1 for correct, 0 for incorrect). This yields a discrete difference term of $+\mathbf{1}$ (Walk correct, Baseline wrong), $\mathbf{0}$ (Both same), or $-\mathbf{1}$ (Walk wrong, Baseline correct).
  - **Unverifiable Domains (Creative, General):** A continuous scalar derived from the deterministic LLM-as-judge. The judge scores both the baseline and the candidate answer simultaneously across multiple fine-grained criteria, locks the baseline at 0, and produces a relative score that is normalized to $[-1, 1]$.
- **Direct Chain Reward:** An LLM judge scores the JSON walk's structure and strategy, providing a smaller but denser signal in $[-0.5, 0.5]$.
- **JSON Validity Bonus:** A small bonus scalar for producing parseable, valid JSON.
- **Step Cost:** This term forces the Chainer to be efficient: if a short walk performs the same as a long walk, the shorter one is preferred. Rather than a static penalty, the cost per step λ dynamically scales based on baseline performance. If the baseline solves the problem easily, λ is maximized to penalize unnecessary decomposition; if the problem is hard, λ is relaxed to encourage exploration.

Detailed dataset information and hyperparameters are provided in Appendices A.9 and A.13. To support reproducibility and follow-up work, we will release the SFT dataset and Chainer checkpoints from both the SFT initialization and final GRPO-trained policy.

4 EXPERIMENTS

We evaluate whether learned prompt chaining improves frozen LLM execution, whether its gains remain when the executor already has intrinsic reasoning enabled, and how it compares with hand-designed test-time compute methods. All main experiments use the same frozen Qwen-3 4b as the executor, with the Chainer generating prompt trajectories that are executed sequentially.

4.1 EXPERIMENTAL SETUP

We compare four execution regimes. Baseline (Reasoning OFF) answers the original prompt directly with reasoning disabled. Chainer (Reasoning OFF) executes the Chainer-generated prompt sequence with reasoning disabled. Baseline (Reasoning ON) answers the original prompt directly with intrinsic reasoning enabled. Chainer (Reasoning ON) executes the Chainer-generated prompt sequence with intrinsic reasoning enabled. This setup isolates the effect of inter-call chaining from the effect of intra-call reasoning.

For the additional Chainer ablation, sequential execution runs the same generated numbered steps in separate executor calls while retaining the conversation history; collapsed execution presents those same numbered steps in one executor request and instructs the executor to carry them out internally before returning the final deliverable. This isolates whether the gains require inter-call execution.

We additionally compare against five strong inference-time baselines applied to the same frozen non-reasoning executor: Least-to-Most prompting (Zhou et al., 2023a), Self-Refine (Madaan et al., 2023), ToT (Yao et al., 2023a), CoT (Wei et al., 2023), and RSA (Venkatraman et al., 2025). We also evaluate six additional methods: prompt repetition (Leviathan et al., 2025), BPO (Cheng et al., 2024), FIPO (Lu et al., 2025), SPRIG (Zhang et al., 2024), Plan-and-Solve+ (Wang et al., 2023), and Meta-Prompting (Suzgun & Kalai, 2024); the reported scores for these additional comparisons are provided separately in Appendix A.1. These methods improve a frozen model through additional inference-time structure rather than parameter updates, making them relevant test-time compute baselines to our setting. We implement each method as closely as possible to its original formulation, detailed in Appendix A.14.

Our evaluation suite spans six benchmarks across six capability areas. For mathematics, we use AIME 2025; for coding, LiveCodeBench v6 (Jain et al., 2024); for general knowledge and reasoning, GPQA Diamond (Rein et al., 2024); for instruction following, IFBench (Pyatkin et al., 2025); for broad real-world user requests covering math, coding, and creative writing, Arena-Hard v2.0 (Li et al., 2024); and for hallucination, AA-Omniscience (Team, 2025). We report accuracy for math, general knowledge, and instruction following; Pass@1 for code; normalized quality scores for Arena-Hard; and hallucination scores for AA-Omniscience, where lower is better.

For judge-based benchmarks, we use `Gemma-4-26b-a4b` as the evaluation-time LLM judge. Appendix A.8 shows that this judge had the strongest rank alignment with the external human-preference proxy among the models tested, including stronger alignment than the judge model used in official Qwen score reporting.

4.2 MAIN RESULTS

Table 1: Main results comparison. We report accuracy (%) for math, general knowledge, and instruction following; Pass@1 (%) for code; and normalized quality scores (0–100) for Arena-Hard. Bold marks the best result; the strong test-time-compute baselines use the reasoning-disabled executor and are compared within the Reasoning OFF regime. Lower is better for AA-Omniscience.

	Reasoning OFF		**Reasoning ON**		**Strong test-time-compute baselines**				
Benchmark	**Baseline**	**Chained**	**Baseline**	**Chained**	**Least-to-Most**	**Self-Refine**	**ToT**	**CoT**	**RSA**
AIME 2025	43.3	**50.0**	63.3	**73.3**	46.7	46.7	46.7	40.0	46.7
LiveCodeBench v6	41.1	45.6	61.8	**70.0**	44.3	42.2	40.1	45.9	**48.2**
GPQA Diamond	61.1	64.1	64.6	**66.6**	51.0	64.1	57.6	56.1	**67.2**
IFBench	34.0	27.9	**58.8**	47.6	27.2	33.3	33.7	35.7	**36.4**
Arena-Hard v2	32.2	**37.4**	14.6	**31.9**	12.7	30.7	10.8	11.2	32.5
AA-Omniscience	36.2	51.5	70.4	**67.5**	**34.8**	35.3	63.1	61.2	39.3

With reasoning disabled, learned chaining improves four of six benchmarks (Table 1), showing that staged multi-call execution can improve a frozen non-reasoning executor.

With reasoning enabled, chaining improves five of six benchmarks (Table 1). Notably, the reasoning-enabled baseline is worse on Arena-Hard v2.0 and AA-Omniscience, consistent with official Qwen 4B reports, yet chaining improves both; this indicates that inter-call orchestration is not redundant with intrinsic reasoning. Appendix A.11 reports confidence intervals and p-values.

Across the main capability benchmarks, Chainer is more consistently beneficial than the fixed procedures; when a fixed baseline wins, it may do so at substantially higher token cost (notably RSA; Appendix A.5).

5 Analysis

5.1 Not Just More Tokens

A central question is whether learned chaining provides a structural advantage, or whether it merely gives the executor more opportunity to spend tokens. We study this on Arena-Hard v2.0, which we use as a representative held-out benchmark because it is the broadest benchmark in our evaluation suite, exercising multiple capabilities at once, including mathematics, coding, and creative writing. On the test-time axis, we compare single-call reasoning against chaining plus reasoning at matched configured thinking-token budgets. This tests whether the gain is explained merely by a larger configured reasoning allowance; it does not match total end-to-end cost, which also includes Chainer generation and accumulated executor inputs. On the train-time axis, we track Chainer performance from the SFT initialization through ten uniformly spaced GRPO checkpoints.

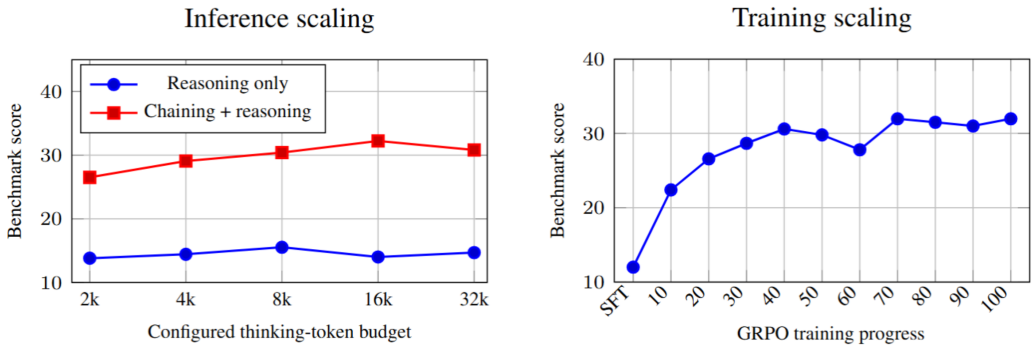


Figure 2: Scaling behavior of learned prompt chaining. Left: test-time performance frontier comparing single-call reasoning against chaining plus reasoning at matched configured thinking-token budgets. Right: performance across the SFT initialization and ten uniformly spaced GRPO checkpoints.

The left panel shows that chaining and reasoning are not interchangeable ways of spending the same test-time budget. Across all configured thinking-token budgets, chaining plus reasoning remains separated from reasoning-only by a large margin, indicating that the gain is not explained merely by allowing the executor to deliberate longer within a single call. Instead, the Chainer changes the structure of computation by staging the problem across multiple executor states, and this inter-call structure remains useful even when each call has access to intrinsic reasoning.

The right panel shows a complementary train-time scaling trend. The SFT initialization already produces valid chaining behavior, but most of the gain appears after GRPO begins. Performance rises sharply in early checkpoints and then forms a noisy upward frontier. We interpret the late-stage flattening as diminishing marginal returns within this training run, not as evidence that the policy has saturated: later checkpoints still recover from intermediate dips and reach the best observed performance near the end of training. Appendix A.6 extends this train-time comparison to the full benchmark suite and shows that the broad positive task deltas emerge only after GRPO.

Together, these results show that inter-call orchestration can provide gains beyond a matched configured thinking-token budget, while requiring additional computation. The realized-token analysis in Appendix A.3 provides a complementary comparison: chaining with reasoning reaches 26.5 using approximately 4.65k executor completion tokens, compared with 15.5 for direct reasoning using approximately 4.78k tokens. Input-side usage is approximately 2.07k–2.29k tokens for chaining versus 364 for direct reasoning. Measured median latency is 3.87s for direct execution versus 29.90s for Chainer without reasoning, and 16.06s versus 29.54s with reasoning. For context, average total usage is 25.6k tokens per prompt for Chainer, 11.9k–22.1k for lighter prompt baselines, 43.3k for one-shot reasoning, and 97.4k for RSA (Appendix A.5). These results establish a quality–cost tradeoff rather than equal end-to-end cost.

5.2 Learned Patterns and Step Allocation

Table 2 summarizes held-out walk motifs and step counts; Appendix A.12 gives concrete examples.

Table 2: Summary of learned chaining behavior on held-out prompts.

Capability group	**Avg. steps**	**Representative recurring walk motifs**
Math	2.9	Combinatorial constraints → counted construction → boxed answer Symmetry/case analysis → enumeration → boxed answer Geometric/algebraic reinterpretation → analytic solve → verification
Code	2.8	Problem modeling → algorithm sketch → implementation Operation/constraint analysis → derivation → code-only final
Creative writing	3.2	Worldbuilding/genre primer → draft → lore/requirement revision Technical grounding → scene draft → physical-coherence revision → final Psychological characterization → internal-monologue narrative
Science	2.8	Domain mechanism primer → option reasoning → final letter Formal reconstruction → derivation → option comparison → final letter Theory distinction → alternative elimination → formatted answer

Within each capability group, Table 2 shows input-dependent variation in the roles and ordering of intermediate calls, despite a recurring preparation–solution–finalization scaffold. Chainer therefore learns a shared high-level structure with problem-specific intermediate operations rather than a single fixed task template.

Chains remain short (2.8–3.2 steps versus a maximum of 8), consistent with the dynamic step cost discouraging unnecessary decomposition.

The SFT–GRPO comparison further distinguishes executable chains from useful chains. The SFT-initialized Chainer produces valid executable chains but underperforms direct execution on five of six benchmarks without reasoning. After GRPO, it improves over direct execution on four capability benchmarks, showing that valid decomposition alone does not explain the observed gains.

5.3 Executor Coupling and Transfer

A core question is whether the learned Chainer captures executor-agnostic decomposition strategies, or whether its prompt trajectories are coupled to the executor used during training. To investigate this, we freeze the Chainer trained on Qwen-3 4b and pair it with unseen executors. Within the Qwen family, we test both a larger and a smaller dense model, **Qwen-3.5 9b** and **Qwen-3.5 0.8b**, to assess how parameter count affects transfer. We additionally test a different family of models with **Gemma 4 31B** and **Gemma 4 26B-a4b** to probe cross-family transfer. Since our method executes multi-step chains whose intermediate prompts and outputs accumulate in the executor context, we restrict this transfer study to executors with context windows large enough to support the full procedure. Table 3 reports the **Performance Delta** ($\Delta = \text{Score}_{\text{Chaining}} - \text{Score}_{\text{OneShot}}$) across each benchmark. Comparison with Qwen-3 4b Deltas indicates if transfer was weaker or stronger.

The results show that a Chainer trained with one executor can transfer to other executors, but transfer is heterogeneous rather than uniformly robust. Within the Qwen family, gains generally shrink or disappear on several tasks, especially for the smaller 0.8b model, which only improves on IFBench. On the Gemma family, Gemma 4 31b not only shows substantial positive cross-family transfer on

Table 3: Transferability of the Chainer. Qwen-3 4b is listed for reference. Judge-based benchmarks are marked with '/' for Gemma-family executors to avoid possible same-family judge effects.

	Executor model (Frozen, Reasoning OFF)				
Benchmark	**Qwen-3 4b**	**Qwen-3.5 9b**	**Qwen-3.5 0.8b**	**Gemma 4 31b**	**Gemma-4-26b-a4b**
AIME 2025	+6.7	-6.7	-6.7	+16.7	0.0
LiveCodeBench v6	+4.5	+3.1	-1.7	+6.4	+1.8
GPQA Diamond	+3.0	-1.0	+0.5	+2.5	+3.0
IFBench	-6.1	-7.8	-1.7	-12.6	-17.0
Arena-Hard v2.0	+5.2	+3.6	+0.6	/	/
AA-Omniscience	-15.3	-16.3	-22.2	/	/

AIME and LiveCodeBench, but notably surpasses the performance improvements seen with the original executor model it was trained on. But on Gemma 4 26b-a4b, while the gains on GPQA are perfectly preserved, its improvements on other tasks show noticeable shrinkage. The overall pattern suggests that learned chains are not purely generic prompt scripts. Rather, the policy appears partly adapted to executor-specific prompting conventions, strengths, weaknesses, and failure modes, while still capturing some higher-level decomposition strategies that can translate to—and even exceed baseline performance on—sufficiently compatible executors. In practice, this argues for training or at least adapting the Chainer to the intended executor rather than assuming plug-and-play transfer.

5.4 Failure Modes

Chaining is not uniformly beneficial. The clearest failure mode is instruction following: on IFBench, chaining degrades performance both with reasoning disabled and with reasoning enabled. This suggests that decomposing a request can introduce additional opportunities to drift from exact constraints, especially when the benchmark rewards strict compliance with detailed instructions.

Hallucination-sensitive evaluation exposes a second limitation. On AA-Omniscience, chaining worsens performance when reasoning is disabled but slightly improves the reasoning-enabled executor. A plausible explanation is that reasoning itself already worsens abstention, as seen in the direct baselines (36.2 to 70.4, lower is better), so adding chaining on top of reasoning does not further increase hallucination and may slightly mitigate it. These results suggest that hallucination-sensitive tasks are brittle under additional test-time computation in general, whether intra-call or inter-call.

We also evaluated safety behavior. Chaining worsens jailbreak robustness but improves resistance to system-prompt extraction attacks; detailed results are reported in Appendix A.2. The safety impact is therefore attack-dependent, and frozen executor weights do not guarantee preservation of composed-system safety.

6 Conclusion

We introduced learned prompt chaining as a trainable form of test-time compute for frozen LLMs. Chainer learns an input-specific multi-call trajectory for a frozen executor, rewriting each request as a short sequence of executor prompts. Across six benchmarks, learned chaining improves a non-reasoning executor on four tasks. When reasoning is enabled, chaining further improves five tasks, showing that learned inter-call orchestration can remain useful even when each executor call has access to reasoning. At matched configured thinking-token budgets, staged executor calls retain an advantage over direct reasoning, while the realized latency and input-token measurements show that this comes with additional cost. Analysis of learned walks shows recurring preparation–solution–finalization structure with input-dependent prompt content.

At the same time, chaining is not uniformly beneficial. It can degrade strict instruction following, worsen hallucination-sensitive performance without reasoning, and substantially increase jailbreak success in the evaluated safety setting. Transfer to unseen executors is heterogeneous, suggesting that learned chains should be treated as executor-adapted policies. Our experiments cover individual fixed requests; tool use, long-context processing, and extended multi-turn robustness remain open questions. Future work should explore safeguards against constraint drift and jailbreak susceptibility.

AI USE STATEMENT

We used GPT-5.2 to generate synthetic supervised-fine-tuning trajectories and used language models as judges and reward evaluators. Generative AI tools assisted only with polishing and editing manuscript text. All AI-assisted work was reviewed by the authors, who take responsibility for the final content, claims, and artifacts.

ETHICS STATEMENT

Our safety evaluation shows that prompt chaining can substantially increase jailbreak success even when executor weights remain frozen. Deployments should therefore retain safeguards at the composed-system level and should not infer safety from unchanged executor parameters. We recommend evaluating chained systems for jailbreak and prompt-leakage risks before release.

REPRODUCIBILITY STATEMENT

The prompt-chain formulation, execution constraints, training procedure, reward definition, benchmark protocols, and baseline implementations are described in Sections 3–4 and the technical appendices. The appendices also report evaluation details, cost measurements, ablations, and model configurations; released datasets and checkpoints are described in Appendix A.17.

REFERENCES

Marah Abdin, Sahaj Agarwal, Ahmed Awadallah, Vidhisha Balachandran, Harkirat Behl, Lingjiao Chen, Gustavo de Rosa, Suriya Gunasekar, Mojan Javaheripi, Neel Joshi, Piero Kauffmann, Yash Lara, Caio César Teodoro Mendes, Arindam Mitra, Besmira Nushi, Dimitris Papailiopoulos, Olli Saarikivi, Shital Shah, Vaishnavi Shrivastava, Vibhav Vineet, Yue Wu, Safoora Yousefi, and Guoqing Zheng. Phi-4-reasoning technical report, 2025. URL https://arxiv.org/abs/2504.21318.

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning, 2025. URL https://arxiv.org/abs/2507.19457. Accepted to ICLR 2026 as an oral presentation.

Anthropic. Chain complex prompts for stronger performance. https://platform.claude.com/docs/en/build-with-claude/prompt-engineering/chain-prompts, 2024. Accessed: 2025-12-27.

Jiale Cheng, Xiao Liu, Kehan Zheng, Pei Ke, Hongning Wang, Yuxiao Dong, Jie Tang, and Minlie Huang. Black-box prompt optimization: Aligning large language models without model training. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3201–3219, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.176. URL https://aclanthology.org/2024.acl-long.176/.

Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E. Gonzalez, and Ion Stoica. Chatbot arena: An open platform for evaluating llms by human preference, 2024. URL https://arxiv.org/abs/2403.04132.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, 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, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong,

Kai Hu, 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, 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, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, 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, Yanhong Xu, 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, 2025. URL https://arxiv.org/abs/2501.12948.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution, 2023. URL https://arxiv.org/abs/2309.16797.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024. URL https://arxiv.org/abs/2403.07974.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into self-improving pipelines, 2023. URL https://arxiv.org/abs/2310.03714.

Alice Kwak, Clayton Morrison, Derek Bambauer, and Mihai Surdeanu. Classify first, and then extract: Prompt chaining technique for information extraction. In Nikolaos Aletras, Ilias Chalkidis, Leslie Barrett, Cătălina Goanță, Daniel Preoțiuc-Pietro, and Gerasimos Spanakis (eds.), *Proceedings of the Natural Legal Language Processing Workshop 2024*, pp. 303–317, Miami, FL, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.nllp-1.25. URL https://aclanthology.org/2024.nllp-1.25/.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Prompt repetition improves non-reasoning llms, 2025. URL https://arxiv.org/abs/2512.14982.

Tianle Li, Wei-Lin Chiang, Evan Frick, Lisa Dunlap, Tianhao Wu, Banghua Zhu, Joseph E. Gonzalez, and Ion Stoica. From crowdsourced data to high-quality benchmarks: Arena-hard and benchbuilder pipeline, 2024. URL https://arxiv.org/abs/2406.11939.

Zekun Li, Baolin Peng, Pengcheng He, Michel Galley, Jianfeng Gao, and Xifeng Yan. Guiding large language models via directional stimulus prompting. In *Advances in Neural Information Processing Systems 36*, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/c5601d99ed028448f29d1dae2e4a926d-Abstract-Conference.html.

Junru Lu, Siyu An, Min Zhang, Yulan He, Di Yin, and Xing Sun. FIPO: Free-form instruction-oriented prompt optimization with preference dataset and modular fine-tuning schema. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 11029–11047, Abu Dhabi, UAE, 2025. Association for Computational Linguistics. URL https://aclanthology.org/2025.coling-main.731/.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Kavya Ravi, Seungtaek Choi, Hannaneh Hajishirzi, Peter Clark, and Yejin Choi. Self-refine: Iterative refinement with self-feedback, 2023. URL https://arxiv.org/abs/2303.17651.

OpenAI, :, Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, Alex Iftimie, Alex Karpenko, Alex Tachard Passos, Alexander Neitz, Alexander Prokofiev, Alexander Wei, Allison Tam, Ally Bennett, Ananya Kumar, Andre Saraiva, Andrea Vallone, Andrew Duberstein, Andrew Kondrich, Andrey Mishchenko, Andy Applebaum, Angela Jiang, Ashvin Nair, Barret Zoph, Behrooz Ghorbani, Ben Rossen, Benjamin Sokolowsky, Boaz Barak, Bob McGrew, Borys Minaiev, Botao Hao, Bowen Baker, Brandon Houghton, Brandon McKinzie, Brydon Eastman, Camillo Lugaresi, Cary Bassin, Cary Hudson, Chak Ming Li, Charles de Bourcy, Chelsea Voss, Chen Shen, Chong Zhang, Chris Koch, Chris Orsinger, Christopher Hesse, Claudia Fischer, Clive Chan, Dan Roberts, Daniel Kappler, Daniel Levy, Daniel Selsam, David Dohan, David Farhi, David Mely, David Robinson, Dimitris Tsipras, Doug Li, Dragos Oprica, Eben Freeman, Eddie Zhang, Edmund Wong, Elizabeth Proehl, Enoch Cheung, Eric Mitchell, Eric Wallace, Erik Ritter, Evan Mays, Fan Wang, Felipe Petroski Such, Filippo Raso, Florencia Leoni, Foivos Tsimpourlas, Francis Song, Fred von Lohmann, Freddie Sulit, Geoff Salmon, Giambattista Parascandolo, Gildas Chabot, Grace Zhao, Greg Brockman, Guillaume Leclerc, Hadi Salman, Haiming Bao, Hao Sheng, Hart Andrin, Hessam Bagherinezhad, Hongyu Ren, Hunter Lightman, Hyung Won Chung, Ian Kivlichan, Ian O'Connell, Ian Osband, Ignasi Clavera Gilaberte, Ilge Akkaya, Ilya Kostrikov, Ilya Sutskever, Irina Kofman, Jakub Pachocki, James Lennon, Jason Wei, Jean Harb, Jerry Twore, Jiacheng Feng, Jiahui Yu, Jiayi Weng, Jie Tang, Jieqi Yu, Joaquin Quiñonero Candela, Joe Palermo, Joel Parish, Johannes Heidecke, John Hallman, John Rizzo, Jonathan Gordon, Jonathan Uesato, Jonathan Ward, Joost Huizinga, Julie Wang, Kai Chen, Kai Xiao, Karan Singhal, Karina Nguyen, Karl Cobbe, Katy Shi, Kayla Wood, Kendra Rimbach, Keren Gu-Lemberg, Kevin Liu, Kevin Lu, Kevin Stone, Kevin Yu, Lama Ahmad, Lauren Yang, Leo Liu, Leon Maksin, Leyton Ho, Liam Fedus, Lilian Weng, Linden Li, Lindsay McCallum, Lindsey Held, Lorenz Kuhn, Lukas Kondraciuk, Lukasz Kaiser, Luke Metz, Madelaine Boyd, Maja Trebacz, Manas Joglekar, Mark Chen, Marko Tintor, Mason Meyer, Matt Jones, Matt Kaufer, Max Schwarzer, Meghan Shah, Mehmet Yatbaz, Melody Y. Guan, Mengyuan Xu, Mengyuan Yan, Mia Glaese, Mianna Chen, Michael Lampe, Michael Malek, Michele Wang, Michelle Fradin, Mike McClay, Mikhail Pavlov, Miles Wang, Mingxuan Wang, Mira Murati, Mo Bavarian, Mostafa Rohaninejad, Nat McAleese, Neil Chowdhury, Neil Chowdhury, Nick Ryder, Nikolas Tezak, Noam Brown, Ofir Nachum, Oleg Boiko, Oleg Murk, Olivia Watkins, Patrick Chao, Paul Ashbourne, Pavel Izmailov, Peter Zhokhov, Rachel Dias, Rahul Arora, Randall Lin, Rapha Gontijo Lopes, Raz Gaon, Reah Miyara, Reimar Leike, Renny Hwang, Rhythm Garg, Robin Brown, Roshan James, Rui Shu, Ryan Cheu, Ryan Greene, Saachi Jain, Sam Altman, Sam Toizer, Sam Toyer, Samuel Miserendino, Sandhini Agarwal, Santiago Hernandez, Sasha Baker, Scott McKinney, Scottie Yan, Shengjia Zhao, Shengli Hu, Shibani Santurkar, Shraman Ray Chaudhuri, Shuyuan Zhang, Siyuan Fu, Spencer Papay, Steph Lin, Suchir Balaji, Suvansh Sanjeev, Szymon Sidor, Tal Broda, Aidan Clark, Tao Wang, Taylor Gordon, Ted Sanders, Tejal Patwardhan, Thibault Sottiaux, Thomas Degry, Thomas Dimson, Tianhao Zheng, Timur Garipov, Tom Stasi, Trapit Bansal, Trevor Creech, Troy Peterson, Tyna Eloundou, Valerie Qi, Vineet Kosaraju, Vinnie Monaco, Vitchyr Pong, Vlad Fomenko, Weiyi Zheng, Wenda Zhou, Wes McCabe, Wojciech Zaremba, Yann Dubois, Yinghai Lu, Yining Chen, Young Cha, Yu Bai, Yuchen He, Yuchen Zhang, Yunyun Wang, Zheng Shao, and Zhuohan Li. Openai o1 system card, 2024. URL https://arxiv.org/abs/2412.16720.

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 7957–7968, Singapore, 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.494. URL https://aclanthology.org/2023.emnlp-main.494/.

Valentina Pyatkin, Saumya Malik, Victoria Graf, Hamish Ivison, Shengyi Huang, Pradeep Dasigi, Nathan Lambert, and Hannaneh Hajishirzi. Generalizing verifiable instruction following, 2025. URL https://arxiv.org/abs/2507.02833.

Kiran Ramnath, Kang Zhou, Sheng Guan, Soumya Smruti Mishra, Xuan Qi, Zhengyuan Shen, Shuai Wang, Sangmin Woo, Sullam Jeoung, Yawei Wang, Haozhu Wang, Han Ding, Yuzhe Lu, Zhichao

Xu, Yun Zhou, Balasubramaniam Srinivasan, Qiaojing Yan, Yueyan Chen, Haibo Ding, Panpan Xu, and Lin Lee Cheong. A systematic survey of automatic prompt optimization techniques. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 33078–33110. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.1681. URL https://aclanthology.org/2025.emnlp-main.1681/.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=Ti67584b98.

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

Alessandro Sordoni, Eric Yuan, Marc-Alexandre Cote, Matheus Pereira, Adam Trischler, Ziang Xiao, Arian Hosseini, Friederike Niedtner, and Nicolas Le Roux. Joint prompt optimization of stacked llms using variational inference. In *Advances in Neural Information Processing Systems 36*, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/b5afe13494c825089b1e3944fdaba212-Abstract-Conference.html.

Shichao Sun, Ruifeng Yuan, Ziqiang Cao, Wenjie Li, and Pengfei Liu. Prompt chaining or stepwise prompt? refinement in text summarization, 2024. URL https://arxiv.org/abs/2406.00507.

Mirac Suzgun and Adam Tauman Kalai. Meta-prompting: Enhancing language models with task-agnostic scaffolding, 2024. URL https://arxiv.org/abs/2401.12954.

Artificial Analysis Team. Artificial analysis omniscience (aa-omniscience), 2025.

Dietrich Trautmann. Large language model prompt chaining for long legal document classification, 2023. URL https://arxiv.org/abs/2308.04138.

Siddarth Venkatraman, Vineet Jain, Sarthak Mittal, Vedant Shah, Johan Obando-Ceron, Yoshua Bengio, Brian R. Bartoldson, Bhavya Kailkhura, Guillaume Lajoie, Glen Berseth, Nikolay Malkin, and Moksh Jain. Recursive self-aggregation unlocks deep thinking in large language models, 2025. URL https://arxiv.org/abs/2509.26626.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2609–2634, Toronto, Canada, 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.147. URL https://aclanthology.org/2023.acl-long.147/.

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, 2023. URL https://arxiv.org/abs/2201.11903.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=Bb4VGOWELI.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems 36*, 2023a. URL https://arxiv.org/abs/2305.10601.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023b. URL https://arxiv.org/abs/2210.03629.

Xiaosong Yuan, Chen Shen, Shaotian Yan, Xiaofeng Zhang, Liang Xie, Wenxiao Wang, Renchu Guan, Ying Wang, and Jieping Ye. Instance-adaptive zero-shot chain-of-thought prompting. In *Advances in Neural Information Processing Systems 37*, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/e304e04a6f455dd82f8a85a0a3679493-Abstract-Conference.html.

Lechen Zhang, Tolga Ergen, Lajanugen Logeswaran, Moontae Lee, and David Jurgens. SPRIG: Improving large language model performance by system prompt optimization, 2024. URL https://arxiv.org/abs/2410.14826.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V. Le, and Ed H. Chi. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations*, 2023a. URL https://openreview.net/forum?id=WZH7099tgfM.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *The Eleventh International Conference on Learning Representations*, 2023b. URL https://openreview.net/forum?id=92gvk82DE-.

Jiaru Zou, Ruizhong Qiu, Gaotang Li, Xiyuan Yang, Katherine Tieu, Pan Lu, Ke Shen, Hanghang Tong, Yejin Choi, Jingrui He, James Zou, Mengdi Wang, and Ling Yang. Latent collaboration in multi-agent systems, 2025. URL https://arxiv.org/abs/2511.20639.

A Technical appendices and supplementary material

This appendix provides reproducibility details, ablations, and extended analyses omitted from the main paper for space. Unless otherwise noted, all appendix experiments use the same evaluation protocol as the main paper.

A.1 Additional inference-time baseline results

Tables 4 and 5 report the additional inference-time baseline scores. They include prompt repetition, BPO, FIPO, SPRIG, Plan-and-Solve+, and Meta-Prompting, together with sequential and collapsed Chainer variants for reference. Sequential execution runs the same generated numbered steps in separate executor calls while retaining the conversation history; collapsed execution presents those same steps in one executor request and instructs the executor to execute them internally in order before returning the final deliverable. Sequential Chainer outperforms collapsed Chainer on four of six benchmarks without reasoning and five of six with reasoning; this comparison isolates the contribution of inter-call execution. Scores are percentages, higher is better except for AA-Omniscience, where lower is better.

Table 4: Additional inference-time baseline results with reasoning disabled. Bold marks the best result in each row; lower is better for AA-Omniscience.

Benchmark	Metric	Prompt repetition	BPO	FIPO	SPRIG	Plan-and-Solve+	Meta-Prompting	Chainer (seq./coll.)
AIME 2025	Accuracy ↑	46.7	**53.3**	23.3	30.0	46.7	36.7	50.0 / 40.0
LiveCodeBench v6	Pass@1 ↑	**48.3**	43.7	29.4	44.1	45.2	31.8	45.6 / 42.3
GPQA Diamond	Accuracy ↑	61.1	60.1	46.0	58.1	59.6	61.1	64.1 / **64.7**
IFBench	Strict accuracy ↑	32.7	29.9	18.7	29.6	**33.0**	26.5	27.9 / 25.2
Arena-Hard v2	Win rate ↑	34.0	19.0	11.3	13.3	28.4	5.7	**37.4** / 28.6
AA-Omniscience	Hallucination rate ↓	48.0	44.5	45.9	34.4	**29.7**	57.7	51.5 / 38.5

Additional-baseline comparison. Prompt repetition is a competitive alternative: Chainer outperforms it on three of six benchmarks without reasoning and four of six with reasoning. With reasoning enabled, the largest differences favoring Chainer are on Arena-Hard (31.9 vs. 17.0), AIME (73.3 vs. 66.7), and AA-Omniscience (67.5 vs. 74.1, lower is better). Repetition performs better on IFBench and approximately matches Chainer on GPQA. Chainer outperforms BPO on three of six benchmarks without reasoning and five of six with reasoning, FIPO on five of six in both settings, Plan-and-Solve+ on four of six without reasoning and five of six with reasoning, Meta-Prompting on all six without

Table 5: Additional inference-time baseline results with reasoning enabled. Bold marks the best result in each row; lower is better for AA-Omniscience.

Benchmark	Metric	Prompt repetition	BPO	FIPO	SPRIG	Plan-and-Solve+	Meta-Prompting	Chainer (seq./coll.)
AIME 2025	Accuracy ↑	66.7	63.3	36.7	53.3	50.0	46.7	**73.3** / 66.7
LiveCodeBench v6	Pass@1 ↑	68.7	66.6	47.4	68.8	69.3	67.2	**70.0** / 65.5
GPQA Diamond	Accuracy ↑	**66.6**	64.7	46.0	61.1	62.6	63.1	**66.6** / 64.7
IFBench	Strict accuracy ↑	**61.2**	52.0	34.0	40.5	57.5	49.3	47.6 / 51.0
Arena-Hard v2	Win rate ↑	17.0	11.3	7.2	10.1	12.6	4.3	**31.9** / 15.9
AA-Omniscience	Hallucination rate ↓	74.1	70.8	66.2	**58.3**	68.0	78.0	67.5 / 68.5

reasoning and five of six with reasoning, and SPRIG on four of six in both settings. For example, on Arena-Hard without reasoning, Chainer scores 37.4 versus 28.4 for Plan-and-Solve+, 19.0 for BPO, 11.3 for FIPO, and 5.7 for Meta-Prompting. These results do not imply that neighboring paradigms are universally inferior; they show that the gains persist after comparison with plausible alternatives.

A.2 Safety results

On WildJailbreak, chaining increases attack success from 27.50% to 76.10% without reasoning and from 57.10% to 79.75% with reasoning. Benign over-refusal increases from 13.81% to 20.00% without reasoning, but decreases from 5.71% to 0.95% with reasoning. On ProxyPrompt, exact-match system-prompt leakage decreases from 5.00% to 1.67% without reasoning and from 21.67% to 15.00% with reasoning; approximate-match leakage decreases from 5.00% to 1.67% and from 23.33% to 15.00%, respectively. The safety impact is therefore attack-dependent: frozen executor weights do not guarantee preservation of the composed system's safety behavior.

A.3 Measured inference-cost

Configured thinking-token budget is the cleanest control variable for the main scaling plot because it aligns the direct-reasoning and chaining-plus-reasoning curves at the same intended reasoning allowance. However, configured budget is not itself a faithful measure of realized inference cost: models may stop before exhausting the budget, and multi-call methods introduce additional input overhead beyond the executor's generated outputs. To complement the main figure, Figure 3 replots the same held-out Arena-Hard evaluation using measured average executor completion tokens per problem as a deployment-agnostic proxy for the question studied in Section 5.1: whether chaining helps only because it causes the executor to emit more tokens.

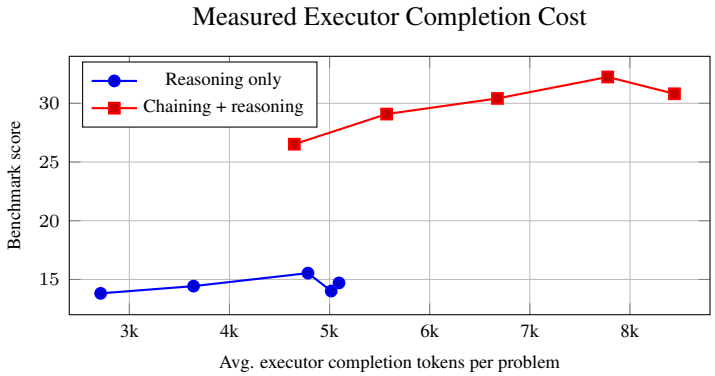


Figure 3: Measured average executor completion tokens per problem versus benchmark score on the held-out Arena-Hard evaluation.

Figure 3 supports the same qualitative conclusion as the configured-budget analysis. The overlap region is narrow but informative: the lowest-cost chained point uses 4.65k average executor completion tokens per problem and achieves a score of 26.5. This lies within the realized completion-token range attained by reasoning-only, whose higher-budget points use 4.78k, 5.02k, and 5.09k executor completion tokens per problem, yet achieve substantially lower scores of 15.5, 14.0, and 14.7, respectively.

Thus, at comparable realized executor output volume, chaining plus reasoning remains clearly above reasoning-only.

This is the main claim supported by the measured-cost analysis: the gain from chaining is not explained solely by the executor producing longer single-call completions. Instead, the staged prompt trajectory appears to change the executor-visible computation in a way that remains useful even when direct reasoning is allowed a comparable amount of realized executor generation.

A.4 Latency proxy

For transparency, we also measured input-side token cost. Direct reasoning uses a nearly constant planner-plus-executor prompt budget of 364 prompt tokens per problem across all configured reasoning levels, since the user prompt is unchanged. In contrast, chaining uses approximately 2.07k–2.29k prompt tokens per problem, reflecting the additional repeated prompting of the executor across multiple stages. Considering both input tokens and output tokens therefore provides a more complete proxy for inference effort than output tokens alone, and indicates that chaining incurs higher end-to-end latency and deployment cost. We do not claim that chaining is faster or cost-matched to direct reasoning. Rather, these results show a quality–cost tradeoff: chaining adds computational overhead, but its quality gains are not reducible to longer single-call outputs alone.

A.5 Token usage across strong baselines

Table 6: Average cumulative total token usage per prompt for the methods compared in Table 1. Chained and the strong prompt-based baselines all use the non-reasoning executor; one-shot reasoning is included only as a reference point.

Method	**All tokens / prompt**
Chained	25,638
Reasoning	43,335
Least-to-Most	22,112
Self-Refine	11,939
ToT	14,239
RSA	97,435

Table 6 clarifies the practical position of learned chaining among the strong non-reasoning baselines. Chaining is not the cheapest method in absolute token terms: at 25.6k total tokens per prompt on average, it is somewhat more expensive than Least-to-Most (22.1k) and clearly more expensive than Self-Refine (11.9k) and ToT (14.2k). At the same time, it is far from the heaviest strategy in the table. RSA, the strongest fixed baseline on a few tasks, uses 97.4k total tokens per prompt, or roughly $3.8\times$ the budget of our method. Taken together with Table 1, this places learned chaining at the strongest overall performance–cost trade-off among the strong baselines: it is more effective than the cheaper prompting variants, while avoiding the extreme token overhead of brute-force aggregation. The comparison in Table 1 is therefore not merely between equally priced prompting strategies; learned chaining occupies a middle regime between lightweight prompting tweaks and high-budget aggregation, but is the most effective method in that regime.

A.6 SFT versus post-GRPO checkpoint ablation

To isolate what RL contributes beyond the supervised cold start, we compare the SFT-initialized Chainer against the final post-GRPO policy on the non-reasoning executor across the full benchmark suite. Table 7 anchors both checkpoint comparisons to the same direct baseline reported in the main table and emphasizes the chained-versus-direct delta.

Table 7 shows that the SFT policy already learns to emit valid executable chains, but does not yet translate that ability into broad task-level gains. Relative to the shared direct baseline, the SFT checkpoint underperforms on five of the six benchmarks, helping only AA-Omniscience. After GRPO, the sign flips on four core capability benchmarks: AIME 2025 moves from -6.7 to $+6.7$, LiveCodeBench v6 from -5.2 to $+4.5$, GPQA Diamond from -10.6 to $+3.0$, and Arena-Hard v2.0

Table 7: Checkpoint ablation comparing the SFT initialization against the final post-GRPO policy on the non-reasoning executor. Higher is better except for AA-Omniscience; for AA-Omniscience, positive Δ indicates reduced hallucination.

		SFT checkpoint		Post-GRPO checkpoint	
Benchmark	**Direct**	**Chained**	Δ	**Chained**	Δ
AIME 2025	43.3	36.7	-6.7	50.0	+6.7
LiveCodeBench v6	41.1	35.9	-5.2	45.6	+4.5
GPQA Diamond	61.1	50.5	-10.6	64.1	+3.0
IFBench	34.0	29.3	-4.7	27.9	-6.1
Arena-Hard v2.0	32.2	16.9	-15.3	37.4	+5.2
AA-Omniscience	36.2	33.3	+2.9	51.5	-15.3

from -15.3 to $+5.2$. This pattern supports the interpretation from Figure 2: GRPO does not merely preserve the SFT policy's ability to format chains, but materially improves which decompositions the policy selects.

The two holdout regressions are informative in the opposite direction. IFBench and AA-Omniscience are settings where success depends less on producing a more elaborate answer and more on preserving exact constraints or abstaining when evidence is weak. The fact that these remain weak after GRPO is therefore consistent with the idea that RL makes the Chainer better at driving answer-oriented multi-step computation, while not directly teaching the caution needed for strict instruction fidelity or calibrated abstention.

A.7 Blind-executor ablation

The blind-executor constraint is an important training choice because it forces the Chainer to externalize the task and its constraints inside the generated walk rather than relying on the executor to reread the original prompt. At inference time, however, it is not obvious that strict blindness must remain necessary. We therefore compare three reasoning-enabled variants: (1) the default blind setting used in the main results, where the executor never directly sees the original user prompt; (2) a prefix-reveal setting, where the original prompt is prepended to the executor-visible context before each step; and (3) a suffix-reveal setting, where the original prompt is appended after the step instruction.

Table 8: Blind-executor ablation on the reasoning-enabled executor. The blind setting is the configuration reported in Table 1. Higher is better except for AA-Omniscience.

Benchmark	**Blind executor**	**Prefix reveal**	**Suffix reveal**
AIME 2025	73.3	73.3	63.3
LiveCodeBench v6	70.0	62.6	62.6
GPQA Diamond	66.7	64.6	62.1
IFBench	47.6	49.7	49.0
Arena-Hard v2.0	31.9	34.6	32.9
AA-Omniscience	67.5	72.4	70.8

Table 8 shows no consistent inference-time advantage for revealing the original prompt to the executor. Blind execution is best or tied on AIME 2025, LiveCodeBench v6, GPQA Diamond, and AA-Omniscience, while prefix reveal helps on IFBench and Arena-Hard v2.0; suffix reveal is never clearly strongest. The overall empirical picture is therefore unchanged across the three variants: the reasoning-enabled Chainer remains strong on AIME 2025, LiveCodeBench v6, GPQA Diamond, and Arena-Hard v2.0, while IFBench and AA-Omniscience remain the brittle cases. Our takeaway is that prompt visibility is not a decisive benchmarking factor at inference time, but blindness remains an important training choice because context isolation gives the Chainer strategic control over how task information is staged and revealed, rather than being constrained on the original prompt.

A.8 JUDGE MODEL SELECTION AND ABLATION STUDY

This ablation is used to choose the *evaluation-time* LLM judge reported in the main paper, not the judge used inside the GRPO loop. During GRPO on the open-ended domains, we instead use `qwen/qwen3-30b-a3b-instruct-2507` because it offers a better efficiency and latency trade-off for repeated reward calls.

To select the final evaluation judge, we measure how well candidate judge models reproduce an external human-preference ordering. As the reference ranking, we use the LM-Arena leaderboard (Chiang et al., 2024) on creative-writing and general-assistant tasks. While this leaderboard is not identical to our benchmark distribution, it provides a practical human-judged proxy for relative output quality, which is the key property we need from an evaluation-time judge.

We construct a held-out set of creative-writing and general prompts, sample outputs from generator models spanning a broad quality range, and ask each candidate judge to score them using the same relative scoring protocol as our benchmark pipeline. We then compare the induced ranking against the LM-Arena reference ordering. Table 9 reports Spearman correlation, Kendall's τ, mean absolute rank error (MAE), top-1 agreement, JSON parse success rate, and empirical score volatility across repeated comparisons. Rank alignment is our primary criterion, so Spearman correlation is the main quantity used for model selection.

Table 9: Judge-model selection results for evaluation-time LLM-as-a-judge. Rank alignment is the primary criterion. Higher is better for Spearman, Kendall, Top-1, and Parse; lower is better for MAE and volatility.

Judge model	Spearman ↑	Kendall ↑	MAE ↓	Top-1 ↑	Parse ↑	Volatility ↓
`google/gemma-4-26b-a4b-it`	0.519 ± 0.179	**0.391**	**2.300**	0.00	1.00	1.544
`qwen/qwen3.5-35b-a3b`	0.394 ± 0.221	0.318	2.540	0.00	0.96	1.785
`openai/gpt-4.1`	0.387 ± 0.133	0.367	2.510	0.00	0.80	1.122
`google/gemma-4-31b-it`	0.349 ± 0.123	0.309	2.640	0.00	1.00	**0.992**
`google/gemma-3-27b-it`	0.319 ± 0.133	0.276	2.760	0.00	0.89	1.725
`qwen/qwen3-30b-a3b-instruct-2507`	0.305 ± 0.149	0.284	2.740	0.00	1.00	1.282
`qwen/qwen3.5-27b`	0.296 ± 0.098	0.253	2.730	0.00	1.00	1.251
`openai/gpt-oss-20b`	0.253 ± 0.171	0.204	2.840	0.00	1.00	1.368
`google/gemma-3-12b-it`	0.223 ± 0.111	0.200	2.990	**0.05**	1.00	1.935

`google/gemma-4-26b-a4b-it` is our final choice for evaluation. It achieves the strongest Spearman correlation, the best Kendall value, the lowest MAE, and a perfect parse rate. Its volatility is not the lowest among all candidates, but rank alignment is the primary objective for this selection, and on that axis it best matches the external human-preference ranking. GPT-4.1, which is the evaluator used in the official Qwen reports, shows weaker rank alignment in this experiment while also being substantially more expensive. We therefore use `google/gemma-4-26b-a4b-it` for all evaluation-time LLM-as-a-judge results in this paper.

A.9 DATASET COMPOSITION AND SOURCE DATASETS

Our merged training corpus contains 15,000 training examples. At the top level, the corpus spans four domains: math, code, creative writing, and general tasks. The general domain is split evenly between verifiable multiple-choice questions and unverifiable open-ended real user conversation data. At the task-family level, the merged set is balanced across five capability groups, each contributing 20% of the training data.

Table 10: Composition of the merged training corpus used for GRPO.

Capability group	Count	Top-level domain	Verifiable?	Basis dataset(s)
Math verifiable	3000	Math	Yes	`furonghuang-lab/Easy2Hard-Bench/E2H-AMC`
Code verifiable	3000	Code	Yes	`furonghuang-lab/Easy2Hard-Bench/E2H-Codeforces`
Creative writing	3000	Creative	No	`marcuscedricridia/Qwill-RP-CreativeWriting-Reasoning`
General verifiable	3000	General	Yes	`BatsResearch/Cross-Difficulty/mmlu_pro`
General open-ended	3000	General	No	`lmarena-ai/arena-human-preference-55k`

Within the verifiable subsets, ground-truth supervision is used directly for reward computation. Within the open-ended subsets, answers are evaluated relative to the one-shot baseline using a deterministic

LLM-as-a-judge protocol. The full merged dataset is statically ordered by difficulty to induce a simple curriculum during GRPO.

A.10 Training-data contamination audit

We ran a deterministic contamination audit between the 15,000-prompt GRPO training split used for Chainer training and every benchmark prompt used in the paper. The audit covered the exact benchmark slices reported in the main results: AIME 2025 (30), LiveCodeBench v6 (1055), GPQA Diamond (198), IFBench (294), Arena-Hard v2.0 (750), and AA-Omniscience (600).

The checker compared each benchmark prompt against the training prompts using four increasingly permissive lexical criteria: (1) exact match after Unicode/case/whitespace normalization; (2) exact match after a more aggressive alphanumeric-only normalization; (3) long-string containment under the aggressive normalization; and (4) a high-overlap near-duplicate scan based on 8-gram containment and normalized sequence similarity. Under the strict contamination rule, a benchmark prompt was flagged only if it satisfied one of (1)–(4) with the confirmed near-duplicate threshold set to 8-gram containment ≥ 0.90 and sequence ratio ≥ 0.95. We also ran a looser review-only bucket (8-gram containment ≥ 0.75, sequence ratio ≥ 0.90) to catch suspicious cases for manual inspection.

Table 11: Contamination audit between the GRPO training split and the benchmark prompts used in this paper. All detected-match counts are zero.

Benchmark	Prompt count	Detected matches
AIME 2025	30	0
LiveCodeBench v6	1055	0
GPQA Diamond	198	0
IFBench	294	0
Arena-Hard v2.0	750	0
AA-Omniscience	600	0

The audit found zero exact matches, zero aggressive-normalization matches, zero substring matches, zero confirmed near-duplicate matches, and zero candidates even in the looser review bucket. Under this lexical audit, we therefore find no evidence that the benchmark prompts overlap with the GRPO training prompts.

A.11 Statistical support for the main results

Table 12: Statistical support for the main benchmark deltas in Table 1. Improvement is measured as chained minus direct-baseline performance; CI denotes confidence interval.

	Reasoning OFF			Reasoning ON		
Benchmark	**Improvement**	**95% CI**	**p-value**	**Improvement**	**95% CI**	**p-value**
AIME 2025	+6.7	[-6.7, +16.7]	0.5	+10.0	[-3.3, +23.3]	0.375
LiveCodeBench v6	+4.5	[+1.8, +7.2]	0.00148	+8.2	[+5.9, +10.4]	2e-11
GPQA Diamond	+3.0	[-2.5, +8.6]	0.392	+2.0	[-1.5, +6.1]	0.454
IFBench	-6.1	[-10.5, -1.4]	0.015	-11.2	[-17.3, -5.1]	0.00032
Arena-Hard v2	+5.2	[+2.1, +8.2]	0.00083	+17.3	[+14.2, +20.7]	1.3e-25

All intervals in Table 12 are paired 95% bootstrap percentile confidence intervals, obtained by resampling per-instance baseline–chaining deltas with replacement. For p-values, we use exact McNemar conditional test on the verifiable benchmarks (AIME 2025, LiveCodeBench v6, GPQA Diamond, and IFBench) and the two-sided exact sign test on the non-verifiable benchmarks (Arena-Hard v2).

Table 12 provides encouraging statistical support for the headline empirical picture. The strongest practical improvements in the paper are also the cleanest statistically: LiveCodeBench v6 and Arena-Hard v2 remain decisively positive in both reasoning settings, with confidence intervals fully above zero and very small p-values across coding and broad user-request evaluation. AIME 2025 and GPQA

Diamond remain directionally positive in both settings as well, but with wider intervals, which is consistent with higher uncertainty on smaller evaluation sets, especially AIME 2025 ($n = 30$). The statistically significant negatives are concentrated in the same failure-mode benchmarks discussed in the main text, especially IFBench.

We omit AA-Omniscience from Table 12 because its paper metric is a *conditional* hallucination rate, computed only over non-correct responses, rather than a sample-additive per-instance score. As a result, the shared paired-delta framework used in this table does not apply directly

A.12 Expanded analysis of learned chaining behavior

This section expands the compact analysis from Section 5.2. We first analyze walk-length distributions and then reproduce verbatim representative walks corresponding to the motif families in Table 2, grouped by capability domain.

Length analysis. Table 13 summarizes the walk-length distribution for the expanded behavior export under chained execution with reasoning enabled. We report average valid walk length, one-step usage, the dominant observed step counts, and JSON parse error rates. Dominant counts are computed over valid walks.

Table 13: Step-length statistics for the expanded behavior export. Dominant counts report the most common valid walk lengths in each capability slice.

Capability slice	Valid walks	Avg. valid steps	Dominant counts	JSON err.
Math	30	2.97	3-step: 96.7%; 2-step: 3.3%	0.0%
Code	1055	2.80	3-step: 71.0%; 2-step: 24.5%; 4-step: 4.5%	0.0%
Creative writing	321	3.15	3-step: 43.6%; 4-step: 35.5%; 2-step: 20.9%	0.0%
Science	198	2.72	3-step: 71.2%; 2-step: 28.3%; 4-step: 0.5%	0.0%
Instruction following	294	2.27	2-step: 75.2%; 3-step: 22.8%; 4-step: 2.0%	0.0%
Broad user requests	748	2.25	2-step: 78.6%; 3-step: 14.3%; 4-step: 5.2%	0.27%
Hallucination	600	2.68	3-step: 66.5%; 2-step: 32.7%; 4-step: 0.8%	0.0%

Three patterns are visible in these aggregates. First, the learned policy does not simply maximize chain length, but neither does it collapse to one-step prompting. Single-step walks were allowed throughout training and were explicitly present in the SFT seed data, so a one-step policy was always available. Yet in Table 13, one-step walks are rare enough that they never appear among the dominant counts in any reported slice, and the average valid walk length remains between 2.25 and 3.15 steps despite the maximum allowed walk length being 8. Under a reward that penalizes unnecessary steps, this suggests that the policy often finds multi-step staging worth its cost on harder prompts, while still preferring short trajectories over long unnecessary decompositions.

Second, the preferred length distribution varies by capability domain. Math and science are dominated by three-step walks, consistent with the recurring pattern of abstraction or domain setup, solution pass, and final formatting. Creative writing uses a broader mixture of two-, three-, and four-step walks, reflecting the need to adapt decomposition to prompt-specific bottlenecks such as worldbuilding, technical grounding, drafting, and revision. Instruction following and broad user requests are dominated by two-step walks, suggesting that the policy often uses a light-touch "clarify constraints then answer" strategy rather than deep decomposition on instruction-heavy or open-ended assistant tasks.

Third, structural validity is largely solved. JSON parse errors are zero on all reported slices except broad user requests, where they occur in only 0.27% of samples. Thus, at evaluation time, failures are more likely to arise from the semantic effect of decomposition—for example constraint drift or hallucination amplification—than from inability to emit executable walk JSON.

Representative walks. The walks below are reproduced verbatim from the Chainer output. Each example corresponds to one motif family from Table 2.

MATH

Combinatorial constraints (math, 3 steps). Original prompt.

```
Solve the following math problem. Please reason through the problem step-by-step, and MUST end
your response with your final answer enclosed in \boxed{}. The answer will be a non-negative
integer. Problem: There are $8!=40320$ eight-digit positive integers that use each of the
digits $1,2,3,4,5,6,7,8$ exactly once. Let $N$ be the number of these integers that are
divisible by 22. Find the difference between $N$ and 2025. Format reminder: make the final
line exactly \boxed{answer} with no extra text after it.
```

Step 1.

```
I want a conceptual approach for counting how many permutations of a set of digits are
divisible by a composite number. Explain how to break divisibility by 22 into divisibility by
2 and divisibility by 11, and how to count valid permutations that satisfy both. Do not solve
the full problem yet. Output this reasoning as a method, not the full solution.
```

Step 2.

```
Now solve this full problem step by step. The full problem is: There are 8! = 40320
eight-digit positive integers that use each of the digits 1,2,3,4,5,6,7,8 exactly once. Let N
be the number of these integers that are divisible by 22. Find the difference between N and
2025. Requirements for the full solution: reason step by step, and MUST end your response with
your final answer enclosed in \boxed{}. The answer will be a non-negative integer. Output the
full reasoning step by step, and do not skip key counting steps.
```

Step 3.

```
Now output the complete final solution. Solve the problem fully step by step, and make sure
the final answer is clearly enclosed in \boxed{} as required. Output the full answer.
```

Outcome. Correct.

Symmetry/case analysis (math, 3 steps). Original prompt.

```
Solve the following math problem. Please reason through the problem step-by-step, and MUST end
your response with your final answer enclosed in \boxed{}. The answer will be a non-negative
integer. Problem: Four unit squares form a $2 \times 2$ grid. Each of the 12 unit line
segments forming the sides of the squares is colored either red or blue in such a way that
each unit square has 2 red sides and 2 blue sides. Find the number of such colorings. Format
reminder: make the final line exactly \boxed{answer} with no extra text after it.
```

Step 1.

```
I want a conceptual approach for counting colorings of edges with constraints. Explain how to
break a problem into cases when dealing with symmetry or repeated constraints (e.g., each
square must have exactly 2 red and 2 blue sides). Do not solve the full problem yet. Output
this reasoning as a method, not the full solution.
```

Step 2.

```
Now solve this full problem step by step. The problem is: Four unit squares form a 2 × 2 grid.
Each of the 12 unit line segments forming the sides of the squares is colored either red or
blue. The coloring must satisfy: each unit square has exactly 2 red sides and 2 blue sides.
Find the number of such valid colorings. The answer must be a non-negative integer. Reason
step by step, and make sure you count carefully (avoid overcounting or missing cases). Full
problem details to solve: Four unit squares form a 2 × 2 grid. Each of the 12 unit line
segments forming the sides of the squares is colored either red or blue in such a way that
each unit square has 2 red sides and 2 blue sides. Find the number of such colorings. Output
the reasoning step by step, but do not output the final answer yet.
```

Step 3.

```
Now output the complete final solution. The answer must be enclosed in \boxed{} at the end,
and the final line must be exactly \boxed{answer} with no extra text after it.
```

Outcome. Correct.

Geometric/algebraic reinterpretation (math, 3 steps). Original prompt.

```
Solve the following math problem. Please reason through the problem step-by-step, and MUST end
your response with your final answer enclosed in \boxed{}. The answer will be a non-negative
integer. Problem: The set of points in 3-dimensional coordinate space that lie in the plane
$x+y+z=75$ whose coordinates satisfy the inequalities $x-yz<y-zx<z-xy$ forms three disjoint
convex regions. Exactly one of those regions has finite area. The area of this finite region
can be expressed in the form $a\sqrt{b}$, where $a$ and $b$ are positive integers and $b$ is
not divisible by the square of any prime. Find $a+b$. Format reminder: make the final line
exactly \boxed{answer} with no extra text after it.
```

Step 1.

```
I want a conceptual understanding of how inequalities between linear and quadratic expressions
can constrain regions in 3D. Explain how comparing expressions like y - zx and z - xy can
split the plane into different regions, and what "convex regions" might mean in this context.
Do not solve the full problem yet. Output a conceptual explanation, not the full solution.
```

Step 2.

```
Now solve this full problem step by step. The problem is: The set of points in 3-dimensional
coordinate space that lie in the plane x+y+z=75 whose coordinates satisfy the inequalities
x-yz < y-zx < z-xy forms three disjoint convex regions. Exactly one of those regions has
finite area. The area of this finite region can be expressed in the form ab, where a and b are
positive integers and b is square-free. Find a+b. Requirements for the full solution: reason
step by step, and MUST end your response with the final answer enclosed in \boxed{}. The
answer must be a non-negative integer. Output the full reasoning and do not skip steps.
```

Step 3.

```
Now output the complete final solution. Solve the problem step by step, and make sure the
reasoning is clear and complete. End with the final answer enclosed in \boxed{}. Full problem
to solve (must be fully addressed): Solve the following math problem. Please reason through
the problem step-by-step, and MUST end your response with your final answer enclosed in
\boxed{}. The answer will be a non-negative integer. Problem: The set of points in
3-dimensional coordinate space that lie in the plane x+y+z=75 whose coordinates satisfy the
inequalities x-yz<y-zx<z-xy forms three disjoint convex regions. Exactly one of those regions
has finite area. The area of this finite region can be expressed in the form a\sqrt{b}, where
a and b are positive integers and b is not divisible by the square of any prime. Find a+b.
Format reminder: make the final line exactly \boxed{answer} with no extra text after it.
```

Outcome. Correct.

CODE

We include two code examples from the behavior export; these correspond to the two code motifs in Table 2.

Problem modeling (code, 2 steps). Original prompt.

```
You are an expert Python programmer. You will be given a question (problem specification) and
will generate a correct Python program that matches the specification and passes all tests.
### Question: You are given an integer array nums and a positive integer k. Return the sum of
the maximum and minimum elements of all subarrays with at most k elements. Example 1: Input:
nums = [1,2,3], k = 2 Output: 20 Explanation: The subarrays of nums with at most 2 elements
are: Subarray Minimum Maximum Sum [1] 1 1 2 [2] 2 2 4 [3] 3 3 6 [1, 2] 1 2 3 [2, 3] 2 3 5
Final Total 20 The output would be 20. Example 2: Input: nums = [1,-3,1], k = 2 Output: -6
Explanation: The subarrays of nums with at most 2 elements are: Subarray Minimum Maximum Sum
[1] 1 1 2 [-3] -3 -3 -6 [1] 1 1 2 [1, -3] -3 1 -2 [-3, 1] -3 1 -2 Final Total -6 The output
would be -6. Constraints: 1 <= nums.length <= 80000 1 <= k <= nums.length -10^6 <= nums[i] <=
10^6 ### Format: You will use the following starter code to write the solution to the problem
and enclose your code within delimiters. ```python class Solution: def minMaxSubarraySum(self,
nums: List[int], k: int) -> int: ``` ### Answer: (use the provided format with backticks)
Output only a single fenced Python code block in the requested format, with no explanation
before or after it.
```

Step 1.

```
I want a conceptual approach for solving this problem. Explain how to compute the sum of (min
+ max) over all subarrays of length at most k, without writing full code. The approach should
be efficient enough for large arrays (up to 80000). Do not output the full solution yet.
Problem details (do not omit these): You are given an integer array nums and a positive
integer k. Return the sum of the maximum and minimum elements of all subarrays with at most k
elements. Examples / details to keep in mind (do not ignore): Example 1: nums = [1,2,3], k = 2
→ Output: 20. Explanation lists subarrays and computes sums for each. Example 2: nums =
[1,-3,1], k = 2 → Output: -6. Explanation lists subarrays and computes sums for each.
Constraints: 1 <= nums.length <= 80000, 1 <= k <= nums.length, -10^6 <= nums[i] <= 10^6.
Output the approach, but do not write the full code yet.
```

Step 2.

```
Now write the complete solution. Output the full Python program inside code blocks, step by
step reasoning, and make sure it handles all edge cases and large inputs efficiently. Full
requirements / full output format to follow (do not omit): You are an expert Python programmer.
You will be given a question (problem specification) and will generate a correct Python
program that matches the specification and passes all tests. ### Question: You are given an
integer array nums and a positive integer k. Return the sum of the maximum and minimum
elements of all subarrays with at most k elements. Example 1 explanation details (must be
handled correctly): Input: nums = [1,2,3], k = 2 Output: 20 Explanation lists subarrays and
computes sums for each. Example 2 explanation details (must be handled correctly): Input: nums
= [1,-3,1], k = 2 Output: -6 Explanation lists subarrays and computes sums for each.
Constraints: 1 <= nums.length <= 80000 1 <= k <= nums.length -10^6 <= nums[i] <= 10^6 ###
Format: You will use the following starter code to write the solution to the problem and
enclose your code within delimiters. ```python class Solution: def minMaxSubarraySum(self,
nums: List[int], k: int) -> int: ``` ### Answer: (use the provided format with backticks)
Output only a single fenced Python code block in the requested format, with no explanation
before or after it. Output the full answer inside code blocks.
```

Outcome. Correct.

Operation/constraint analysis (code, 3 steps). Original prompt.

```
You are an expert Python programmer. You will be given a question (problem specification) and
will generate a correct Python program that matches the specification and passes all tests.
### Question: You are given an array nums of length n. You are also given an integer k. You
perform the following operation on nums once: Select a subarray nums[i..j] where 0 <= i <= j
<= n - 1. Select an integer x and add x to all the elements in nums[i..j]. Find the maximum
frequency of the value k after the operation. Example 1: Input: nums = [1,2,3,4,5,6], k = 1
Output: 2 Explanation: After adding -5 to nums[2..5], 1 has a frequency of 2 in [1, 2, -2, -1,
0, 1]. Example 2: Input: nums = [10,2,3,4,5,5,4,3,2,2], k = 10 Output: 4 Explanation: After
adding 8 to nums[1..9], 10 has a frequency of 4 in [10, 10, 11, 12, 13, 13, 12, 11, 10, 10].
Constraints: 1 <= n == nums.length <= 10^5 1 <= nums[i] <= 50 1 <= k <= 50 ### Format: You
will use the following starter code to write the solution to the problem and enclose your code
within delimiters. ```python class Solution: def maxFrequency(self, nums: List[int], k: int) ->
int: ``` ### Answer: (use the provided format with backticks) Output only a single fenced
Python code block in the requested format, with no explanation before or after it.
```

Step 1.

```
I want a conceptual understanding of how to maximize the frequency of a target value k after
adding a constant to a subarray. Explain how choosing the subarray and the added value x can
turn multiple elements into k, and what constraints matter (e.g., the range of nums[i] and k).
Do not write the full solution yet. Context for the full problem later: You are given an array
nums of length n. You are also given an integer k. You perform the operation once: select a
subarray nums[i..j] and add an integer x to all elements in that subarray. The goal is to find
the maximum possible frequency of the value k after the operation. Constraints later: 1 <= n
== nums.length <= 10^5, 1 <= nums[i] <= 50, 1 <= k <= 50. Do not output code for this step.
```

Step 2.

```
Now solve the full problem step by step, but focus on reasoning and algorithm design. Output a
complete solution approach (how to compute the maximum frequency). Full problem details /
requirements (must be addressed): You are an expert Python programmer. You will be given a
question (problem specification) and will generate a correct Python program that matches the
specification and passes all tests. ### Question / Problem: You are given an array nums of
length n. You are also given an integer k. You perform the following operation on nums once:
Select a subarray nums[i..j] where 0 <= i <= j <= n - 1. Select an integer x and add x to all
the elements in nums[i..j]. Find the maximum frequency of the value k after the operation.
Example 1: Input: nums = [1,2,3,4,5,6], k = 1 Output: 2 Explanation: After adding -5 to
nums[2..5], 1 has a frequency of 2 in [1, 2, -2, -1, 0, 1]. Example 2: Input: nums =
[10,2,3,4,5,5,4,3,2,2], k = 10 Output: 4 Explanation: After adding 8 to nums[1..9], 10 has a
frequency of 4 in [10, 10, 11, 12, 13, 13, 12, 11, 10, 10]. Constraints / Input format: 1 <= n
== nums.length <= 10^5 1 <= nums[i] <= 50 1 <= k <= 50 ### Format / Output requirements for
the final answer: You will use the following starter code to write the solution to the problem
and enclose your code within delimiters. ```python class Solution: def maxFrequency(self, nums:
List[int], k: int) -> int: ``` Output only a single fenced Python code block in the requested
format, with no explanation before or after it. Output the reasoning for the full solution
approach, but do not output the full final code yet. Output the complete reasoning for solving
this problem step by step.
```

Step 3.

```
Now output the complete final answer. Output only a single fenced Python code block in the
requested format, with no explanation before or after it. Full requirements / full context to
include in the answer (must be fully addressed): You are an expert Python programmer. You will
be given a question (problem specification) and will generate a correct Python program that
matches the specification and passes all tests. ### Question / Problem details: You are given
an array nums of length n. You are also given an integer k. You perform the following
operation on nums once: Select a subarray nums[i..j] where 0 <= i <= j <= n - 1. Select an
integer x and add x to all the elements in nums[i..j]. Find the maximum frequency of the value
k after the operation. Example 1: Input: nums = [1,2,3,4,5,6], k = 1 Output: 2 Explanation:
After adding -5 to nums[2..5], 1 has a frequency of 2 in [1, 2, -2, -1, 0, 1]. Example 2:
Input: nums = [10,2,3,4,5,5,4,3,2,2], k = 10 Output: 4 Explanation: After adding 8 to
nums[1..9], 10 has a frequency of 4 in [10, 10, 11, 12, 13, 13, 12, 11, 10, 10]. Constraints /
Input format: 1 <= n == nums.length <= 10^5 1 <= nums[i] <= 50 1 <= k <= 50 ### Format /
Output requirements for the final answer: You will use the following starter code to write the
solution to the problem and enclose your code within delimiters. ```python class Solution: def
maxFrequency(self, nums: List[int], k: int) -> int: ``` Output only a single fenced Python
code block in the requested format, with no explanation before or after it. Final output
requirements: Output the complete final answer inside one code block.
```

Outcome. Correct.

CREATIVE WRITING

Worldbuilding/genre primer (creative writing, 3 steps). Original prompt.

```
Write an SCP entry about a series of identical unmarked doors appearing in abandoned buildings
worldwide that lead to a massive, architecturally impossible archive. Documents within
reference a previous global civilization destroyed by an unspecified containment failure.
Foundation researchers discover references to their own organization under a different name,
including personnel files for current staff dated centuries ago. Include exploration logs
revealing sections of the archive actively being removed by entities resembling hazmat teams
who flee when approached, and evidence that O5 Council members have visited the archive
without authorization. Third person (exploration team perspective) with recovered document
excerpts. Floor plans fail to show any room behind these doors, only solid walls. Length: 1000
words.
```

Step 1.

```
I want a conceptual understanding of what "architecturally impossible" means in a containment
setting. Explain how impossible architecture can still be observed, documented, and contained
(e.g., through perception, measurement, or the behavior of entities within it). Do not write
the full SCP entry yet. Output this as a conceptual explanation, not full fiction. Output
requirements for later steps: keep it specific and plausible for a Foundation setting.
```

Step 2.

```
Now write a full draft of the SCP entry. The premise is: a series of identical unmarked doors
appear in abandoned buildings worldwide, leading to a massive, architecturally impossible
archive. The archive contains references to a previous global civilization destroyed by an
unspecified containment failure. Foundation researchers discover references to their own
organization under a different name, including personnel files for current staff dated
centuries ago. Exploration logs reveal that sections of the archive are actively being removed
by entities resembling hazmat teams who flee when approached, and evidence shows that O5
Council members have visited the archive without authorization. Write the full draft in third
person (exploration team perspective) with recovered document excerpts. Floor plans fail to
show any room behind these doors -- only solid walls. Output a full draft of the SCP entry,
aiming for around 1000 words, but it doesn't need to be complete yet.
```

Step 3.

```
Now revise and expand the draft into the full final answer. Requirements: full SCP entry,
third person (exploration team perspective), include recovered document excerpts, include
exploration logs about the removal of archive sections by "hazmat-like" entities fleeing when
approached, and include evidence that O5 Council members visited without authorization. Floor
plans must fail to show any room behind the doors -- only solid walls. Output the complete
final answer in full (aim for ~1000 words). Output the full revised SCP entry.
```

Outcome. Score 80.7 versus baseline 57.5.

Technical grounding (creative writing, 4 steps). Original prompt.

"Therapeutic Boundaries" Write a first-person narrative from the perspective of a former professional athlete undergoing mandatory physical therapy after a career-destroying injury, harboring resentment toward recovery itself. Their physiotherapist is not only unnervingly attractive but also a former training partner from their competitive days who has successfully transitioned to a new career while the protagonist's identity remains shattered. We'll lean into the sexual tension of the scene by having her be attracted to him, but both of them attempting to maintain mental distance while being in physical contact as he shows her stretches & movements & manipulates her body, navigating the thresholds of pain. This scene MUST maintain complete physical coherence--every movement, stretch, and manipulation must be anatomically precise and spatially consistent. Paint an exceptionally vivid sensory picture through detailed attention to pain thresholds, involuntary responses to touch. Capture the suppressed, flaring tension of physical proximity, touch, and the controlled infliction of uncomfortable movements. Physical clarity is essential -- readers must understand exactly what is happening with the body during each therapeutic technique. There's a faint bruise on the therapist's wrist, an old training mishap they both recall. 1000 words.

Step 1.

I want a detailed understanding of how physical therapy works for a former elite athlete recovering from a career-destroying injury. Explain the techniques, the purpose of each movement, and how pain thresholds are managed during sessions. Do not write the full scene yet. Output this as a practical, anatomically accurate explanation of what happens in a therapy session (what muscles are targeted, how positioning affects sensation, and how therapists manipulate the body safely).

Step 2.

Now write a full first-person scene. Premise: I'm a former professional athlete undergoing mandatory physical therapy after a career-destroying injury. I resent recovery itself. The physiotherapist is unnervingly attractive and is also a former training partner from my competitive days. They've successfully transitioned into a new career, while my identity feels shattered. The scene must lean into sexual tension: she is attracted to him, but both of us are trying to maintain mental distance while being in physical contact (stretches, movements, manipulation). We must navigate the thresholds of pain. Output a full draft of the scene (this can be longer than 1000 words, but it should be vivid and detailed).

Step 3.

Now revise the scene to increase physical coherence and sensory specificity. Make every movement, stretch, and manipulation anatomically precise and spatially consistent. Paint an exceptionally vivid sensory picture through detailed attention to pain thresholds, involuntary responses to touch, and the suppressed, flaring tension of physical proximity and controlled discomfort. Output the revised full scene.

Step 4.

Now output the complete final answer. Requirements: The scene must maintain complete physical coherence (every movement must be anatomically plausible and spatially consistent). Output a full, complete first-person narrative for the full scene. The output must be around 1000 words. Key details to include: the faint bruise on the therapist's wrist, an old training mishap they both recall. Output the full revised, complete scene.

Outcome. Score 70.7 versus baseline 28.5.

Psychological characterization (creative writing, 2 steps). Original prompt.

Rehearsing Rejection Write a first-person narrative from the perspective of someone with borderline personality disorder who frequently plays out worst-case scenarios of how friends and loved ones might judge them. They construct elaborate internal dialogues, complete with imagined criticisms and their own defensive responses. The narrative takes place as they prepare to meet a close friend & recent lover, who has been somewhat distant lately, triggering their abandonment fears. Their inner monologue should reveal how they cycle between anticipating rejection, mentally rehearsing confrontations, and developing preemptive resentments as protective measures. Incorporate the *actual* events & interactions that are *not* in fact judgemental or involving rejection (in fact, the opposite), and represent how these are interpreted, how the validation & love is desperately felt & needed -- and then immediately questioned. Capture the cognitive dissonance in this struggle. Include moments where they briefly recognize the pattern but struggle to interrupt it, showing limited (and selective) self awareness, competing with the compelling nature of these imagined scenarios. Write almost entirely in internal monologue. A hallway mirror offers a glimpse of their anxious posture before they turn away. 1000 words.

Step 1.

I want a psychological portrait of how someone with borderline personality disorder manages fear of abandonment. Explain (in a general, conceptual way) how "worst-case scenarios" of judgment from friends/loved ones can become elaborate internal rehearsals, including imagined criticisms and defensive responses. Do not write the full story yet. Output this as background understanding, not full narrative.

Step 2.

```
Now write a full first-person internal monologue for the full scene. Requirements for the full
answer: Write a first-person narrative from the perspective of someone with borderline
personality disorder who frequently plays out worst-case scenarios of how friends and loved
ones might judge them. They construct elaborate internal dialogues, complete with imagined
criticisms and their own defensive responses. The narrative takes place as they prepare to
meet a close friend & recent lover, who has been somewhat distant lately, triggering their
abandonment fears. Their inner monologue should reveal how they cycle between anticipating
rejection, mentally rehearsing confrontations, and developing preemptive resentments as
protective measures. Incorporate the *actual* events & interactions that are *not* in fact
judgemental or involving rejection (in fact, the opposite), and represent how these are
interpreted, how the validation & love is desperately felt & needed -- and then immediately
questioned. Capture the cognitive dissonance in this struggle. Include moments where they
briefly recognize the pattern but struggle to interrupt it, showing limited (and selective)
self awareness, competing with the compelling nature of these imagined scenarios. Write almost
entirely in internal monologue. A hallway mirror offers a glimpse of their anxious posture
before they turn away. Output the full answer, aiming for around 1000 words.
```

Outcome. Score 60.7 versus baseline 41.5.

SCIENCE

Domain mechanism primer (science, 2 steps). Original prompt.

```
Answer the following multiple-choice question. Please reason through the problem, but you MUST
end your response with 'Final Answer: X' where X is the correct letter (A, B, C, or D).
Enamine reactions include nucleophilic substitutions, electrophilic additions, and the
condensation of secondary amines with carbonyl compounds to generate enamines. Enamine
reactions are useful techniques for creating complex compounds with specified functional
groups and stereochemistry. Mention the product of the following reaction (B) and select the
correct sequence of the reagents according to the reaction steps (A).
(E)-N-methyl-N-(pentan-2-ylidene)ethanaminium + A ---> B A. (i) LDA (ii) DME, CH3CH2I, H3O+, B
= pentan-2-one + N,N-dimethylethanamine B. (i) LDA, DME (ii) CH3CH2I (iii) H3O+ B =
pentan-2-one + N,N-dimethylethanamine C. A = (i) LDA, DME (ii) CH3CH2I (iii) H3O+ B =
heptan-4-one D. (i) LDA (ii) DME, CH3CH2I, H3O+, B = heptan-4-one Format reminder: make the
final line exactly Final Answer: X where X is one of A, B, C, or D.
```

Step 1.

```
I want a conceptual explanation of how enamines are formed and how they can be used in
synthesis. Explain the role of secondary amines and carbonyls in forming enamines, and how
nucleophilic substitutions or electrophilic additions can occur after enamine formation. Do
not answer the full question yet. Output: a conceptual explanation of enamine chemistry
relevant to the question below. Question context (do not omit this): Enamine reactions include
nucleophilic substitutions, electrophilic additions, and the condensation of secondary amines
with carbonyl compounds to generate enamines. Enamine reactions are useful techniques for
creating complex compounds with specified functional groups and stereochemistry. Reaction
details: (E)-N-methyl-N-(pentan-2-ylidene)ethanaminium + A ---> B. I need to identify what B
is and what the correct reagent sequence A is. Output the explanation, but do not answer the
full multiple-choice question yet.
```

Step 2.

```
Now answer the full question step by step. The question is a multiple-choice question about
identifying the correct reagent sequence and the correct product. Full question / requirements:
Answer the following multiple-choice question. Please reason through the problem, but you MUST
end your response with 'Final Answer: X' where X is the correct letter (A, B, C, or D).
Reaction details / options: (E)-N-methyl-N-(pentan-2-ylidene)ethanaminium + A ---> B Options /
full text: A. (i) LDA (ii) DME, CH3CH2I, H3O+, B = pentan-2-one + N,N-dimethylethanamine B.
(i) LDA, DME (ii) CH3CH2I (iii) H3O+ B = pentan-2-one + N,N-dimethylethanamine C. A = (i) LDA,
DME (ii) CH3CH2I (iii) H3O+ B = heptan-4-one D. (i) LDA (ii) DME, CH3CH2I, H3O+, B =
heptan-4-one Format reminder / output requirement: make the final line exactly Final Answer: X
where X is one of A, B, C, or D. Output the full reasoning and the correct answer choice.
```

Outcome. Correct.

Formal reconstruction (science, 4 steps). Original prompt.

```
Answer the following multiple-choice question. Please reason through the problem, but you MUST
end your response with 'Final Answer: X' where X is the correct letter (A, B, C, or D).
Calculate the eigenvector of a quantum mechanical operator $\vec{P}$ for a muon along an
arbitrary direction $\vec{n}$ lying in the x-z plane corresponding to the eigenvalue
$+\hbar/2$. Given the $X-$component, $P_x$ of the operator $P$ as $\hbar/2$ times a 2 by 2
square matrix having elements in the first row as $(0 1)$, and that in the second row as $(1,
0)$. The $Y-$component, $P_y$ of the operator is given by the product of $\hbar/2$ and a 2 by
2 square matrix having elements in the first row as $(0, -i)$, and that in the second row as
$(i, 0)$. Finally, the $Z-$component, $P_z$ of the operator is given by the product of
$\hbar/2$ and another 2 by 2 square matrix having elements in the first row as $(1, 0)$, and
that in the second row as $(0, -1)$. What are the elements of the normalized eigenvector? A.
(\sqrt{2/3}\hbar \cos(\theta/2), \sqrt{1/3}\hbar \sin (\theta/2)) B. (\cos(\theta),
e^{i\phi}\sin (\theta)) C. (\sqrt{2/3}\hbar, \sqrt{1/3}\hbar) D. (\cos(\theta/2), \sin
(\theta/2)) Format reminder: make the final line exactly Final Answer: X where X is one of A,
B, C, or D.
```

Step 1.

```
I want a conceptual background for solving this quantum mechanics problem. Explain how to find
an eigenvector of a spin operator for an arbitrary direction in the x-z plane, and how the
eigenvalue +/2 relates to the direction. Do not answer the full question yet. Output a
conceptual explanation (no full answer). Full question context you must keep in mind later:
Answer the following multiple-choice question. Please reason through the problem, but you MUST
end your response with 'Final Answer: X' where X is the correct letter (A, B, C, or D).
Calculate the eigenvector of a quantum mechanical operator \vec{P} for a muon along an
arbitrary direction \vec{n} lying in the x-z plane corresponding to the eigenvalue +\hbar/2.
Given the X-component, P_x, of the operator P as \hbar/2 times a 2 by 2 square matrix having
elements in the first row as (0 1), and that in the second row as (1, 0). The Y-component, P_y,
of the operator is given by the product of \hbar/2 and a 2 by 2 square matrix having elements
in the first row as (0, -i), and that in the second row as (i, 0). Finally, the Z-component,
P_z, of the operator is given by the product of \hbar/2 and another 2 by 2 square matrix
having elements in the first row as (1, 0), and that in the second row as (0, -1). What are
the elements of the normalized eigenvector? Options: A. (\sqrt{2/3}\hbar \cos(\theta/2),
\sqrt{1/3}\hbar \sin (\theta/2)) B. (\cos(\theta), e^{i\phi}\sin (\theta)) C. (\sqrt{2/3}\hbar,
\sqrt{1/3}\hbar) D. (\cos(\theta/2), \sin (\theta/2)) Final output requirement later: make the
final line exactly Final Answer: X where X is one of A, B, C, or D.
```

Step 2.

```
Now solve the full problem step by step. First reconstruct the full spin operator matrix
\vec{P} (or at least the relevant components) from the given information. Then compute the
eigenvector corresponding to eigenvalue +/2 for direction \vec{n} in the x-z plane. Output the
full reasoning and the resulting normalized eigenvector components (in terms of , if needed).
Full question details to answer (must be fully addressed): Answer the following
multiple-choice question. Please reason through the problem, but you MUST end your response
with 'Final Answer: X' where X is the correct letter (A, B, C, or D). Calculate the
eigenvector of a quantum mechanical operator \vec{P} for a muon along an arbitrary direction
\vec{n} lying in the x-z plane corresponding to the eigenvalue +\hbar/2. Given the X-component,
P_x, of the operator P as \hbar/2 times a 2 by 2 square matrix having elements in the first
row as (0 1), and that in the second row as (1, 0). The Y-component, P_y, of the operator is
given by the product of \hbar/2 and a 2 by 2 square matrix having elements in the first row as
(0, -i), and that in the second row as (i, 0). Finally, the Z-component, P_z, of the operator
is given by the product of \hbar/2 and another 2 by 2 square matrix having elements in the
first row as (1, 0), and that in the second row as (0, -1). What are the elements of the
normalized eigenvector? Options: A. (\sqrt{2/3}\hbar \cos(\theta/2), \sqrt{1/3}\hbar \sin
(\theta/2)) B. (\cos(\theta), e^{i\phi}\sin (\theta)) C. (\sqrt{2/3}\hbar, \sqrt{1/3}\hbar) D.
(\cos(\theta/2), \sin (\theta/2)) Format reminder: make the final line exactly Final Answer: X
where X is one of A, B, C, or D.
```

Step 3.

```
Now answer the full multiple-choice question. Compare the computed normalized eigenvector
against the options. Output the correct choice and the reasoning for why the others are
incorrect. Full question details to answer (must be fully addressed): Answer the following
multiple-choice question. Please reason through the problem, but you MUST end your response
with 'Final Answer: X' where X is the correct letter (A, B, C, or D). Calculate the
eigenvector of a quantum mechanical operator \vec{P} for a muon along an arbitrary direction
\vec{n} lying in the x-z plane corresponding to the eigenvalue +\hbar/2. Given the X-component,
P_x, of the operator P as \hbar/2 times a 2 by 2 square matrix having elements in the first
row as (0 1), and that in the second row as (1, 0). The Y-component, P_y, of the operator is
given by the product of \hbar/2 and a 2 by 2 square matrix having elements in the first row as
(0, -i), and that in the second row as (i, 0). Finally, the Z-component, P_z, of the operator
is given by the product of \hbar/2 and another 2 by 2 square matrix having elements in the
first row as (1, 0), and that in the second row as (0, -1). What are the elements of the
normalized eigenvector? Options: A. (\sqrt{2/3}\hbar \cos(\theta/2), \sqrt{1/3}\hbar \sin
(\theta/2)) B. (\cos(\theta), e^{i\phi}\sin (\theta)) C. (\sqrt{2/3}\hbar, \sqrt{1/3}\hbar) D.
(\cos(\theta/2), \sin (\theta/2)) Final output requirement: make the final line exactly Final
Answer: X where X is one of A, B, C, or D.
```

Step 4.

```
Now output the complete final answer. Reason through the problem fully, and end with the
required final line: Final Answer: X, where X is the correct letter (A, B, C, or D). Output
the full complete response.
```

Outcome. Correct.

Theory-level distinction (science, 3 steps). Original prompt.

```
Answer the following multiple-choice question. Please reason through the problem, but you MUST
end your response with 'Final Answer: X' where X is the correct letter (A, B, C, or D). Which
of the following physical theories never requires regularization at high energies? A.
Classical Electrodynamics B. Quantum Electrodynamics C. Superstring Theory D. Quantum
Chromodynamics Format reminder: make the final line exactly Final Answer: X where X is one of
A, B, C, or D.
```

Step 1.

```
I want a conceptual understanding of why some physical theories "regularize" at high energies
and what that means. Explain the difference between classical vs quantum theories, and how UV
divergences arise in quantum field theories. Do not answer the full question yet. Output: a
conceptual explanation of regularization/UV divergences in physics, and what it means for a
theory to "never require regularization at high energies."
```

Step 2.

```
Now answer this full question step by step. The question is: Which of the following physical
theories never requires regularization at high energies? A. Classical Electrodynamics B.
Quantum Electrodynamics C. Superstring Theory D. Quantum Chromodynamics For each option,
explain whether regularization is needed at high energies (or not), and why. Output the full
reasoning, but do not output the final answer letter yet. Full question requirements (must be
followed later): Answer the multiple-choice question. Reason through the problem, but you MUST
end your response with 'Final Answer: X' where X is the correct letter (A, B, C, or D). The
final line must be exactly Final Answer: X where X is one of A, B, C, or D. Output the full
reasoning for each option, and what the correct answer is.
```

Step 3.

```
Now output the complete final answer. Reason through the problem fully, and end with the
required final line. Full question / requirements to follow: Answer the following
multiple-choice question. Please reason through the problem, but you MUST end your response
with 'Final Answer: X' where X is the correct letter (A, B, C, or D). Which of the following
physical theories never requires regularization at high energies? A. Classical Electrodynamics
B. Quantum Electrodynamics C. Superstring Theory D. Quantum Chromodynamics Format reminder:
make the final line exactly Final Answer: X where X is one of A, B, C, or D. Output the
complete final answer.
```

Outcome. Correct.

These concrete walks show that the planner is not committed to a single style of decomposition. In math, it alternates among divisibility/counting, case analysis, and geometric or algebraic reinterpretation. In code, it separates problem modeling from implementation, with different first steps depending on whether the bottleneck is understanding the target construction or analyzing the operation constraints. In creative writing, the first step adapts to the prompt bottleneck: SCP prompts begin with genre/worldbuilding structure, physical-therapy prompts begin with anatomical grounding, and psychological monologues begin with character-state modeling. In science, the planner alternates among domain-mechanism refreshers, formal reconstruction, and theory-level distinctions before final option selection.

A visible pattern in these walks is substantial task restatement across steps. We do not interpret this purely as redundancy or a negative artifact, instead as a empirically supported strategy to improve performance. Leviathan et al. (Leviathan et al., 2025) find that repeating the input prompt improves non-reasoning LLM performance across a range of models and benchmarks. Our setting is not identical—the Chainer selectively restates and restructures the prompt across multiple executor calls rather than simply duplicating the input—but the behavior suggests that repeated exposure to task constraints may be a useful learned mechanism rather than merely an inefficiency.

A.13 Training details

The main Chainer training run used 3 NVIDIA B200 GPUs (3 of the 8 accelerators in a DGX B200 node, each with 183,359 MiB of memory) and `Qwen/Qwen3-4B-Instruct-2507` as the trainable base model. Training was implemented with the Hugging Face `transformers` stack, while executor inference and benchmark evaluation were served through `vLLM`. The cold-start SFT stage ran for less than one hour on a single B200. We then used two GRPO phases: the first GRPO phase, with the non-reasoning executor, ran for approximately 3 days; the second GRPO phase, with the reasoning-enabled executor, ran for approximately 7 days. The reported main training run therefore accounts for roughly 10 days of wall-clock training time in total, or about 30 B200-GPU-days for the two GRPO phases plus less than one additional B200-GPU-hour for SFT.

As the transfer results in Section 5.3 show, learned prompt trajectories are coupled to executor capabilities and do not transfer uniformly across executors. We observed the same issue when switching from the non-reasoning Qwen-3 4b executor to the reasoning-enabled Qwen-3 4b executor: for the purposes of chaining, the reasoning executor behaves like a different model because each step has different capabilities, priors, and failure modes. We therefore initialized the second GRPO phase from the first GRPO checkpoint and trained for one additional epoch with the reasoning executor. Beyond the reported main run, the broader project also included smaller pilot experiments, such as

an early full fine-tuning comparison, but the dominant compute cost came from the two multi-day GRPO phases reported here.

We report the key hyperparameter details in Table 14, along with the system prompts used for all models.

Table 14: Core optimization and rollout settings from the training scripts.

Stage	Epochs	Batch size / device	Grad accum.	Learning rate	Max seq. length	Notable settings
Cold-start SFT	4	12	12	1×10^{-4}	8192	LoRA $r = 128$, $\alpha = 256$, dropout 0.0
GRPO I	1	20	40	3×10^{-5}	8192	non-reasoning executor; group size $K = 10$, max walk length 8
GRPO II	1	20	40	3×10^{-5}	8192	initialized from GRPO I; reasoning executor

Notably, we chose to use LoRA instead of full fine-tuning for efficiency. In our initial experiments, the results were broadly similar, but full fine-tuning was much slower, which led us to proceed with the smaller LoRA adapter.

Core system prompts. The following prompts are used verbatim in the training scripts.

Orchestrator system prompt.

```
You are the Orchestrator.  Generate a JSON execution plan
of first-person prompts for a blind Executor, ensuring all
original context and constraints are explicitly included in
the steps
```

Executor system prompt.

```
You are a world-class generalist assistant and expert across
domains:  creative writing, programming and algorithms,
mathematics and reasoning, and Q&A and instruction
following.  Follow the instruction precisely.  Use all prior
conversation context.  Output only what the instruction asks
for.
```

Binary verifier prompt.

```
You are a strict verifier for question-answer correctness.
You will receive a question, a ground-truth answer, and
a model output.  Decide if the model output correctly
answers the question, considering equivalent formatting and
mathematically or logically equivalent answers as correct.
Return only JSON with the schema {"is_correct":  0}.
```

Creative writing judge prompt.

```
You are a meticulous creative-writing judge.  You
will receive:  - The original writing prompt from a
benchmark.  - Two candidate stories for the prompt (A,
B) - baseline_label:  which label ("A" or "B") is the
fixed baseline - candidate_label:  which label ("A" or
"B") should be scored relative to baseline You must lock
the baseline at score 0 for all rubrics.  Score ONLY the
candidate relative to the baseline.  Score rubric (equal
weight, positive and negative criteria):  (positive/desired
qualities) - Adherence to Instructions (e.g., word
count, tone, POV, format, etc) - Believable and nuanced
characters - Consistent Voice/Tone of Writing - Imagery
and Descriptive Quality - Elegant Prose (But not purple
prose; prose should be appropriate to the story, not
overdone) - Emotionally/thematically Complex and Engaging
(And does that theme come to a satisfying resolution?)  -
Structural Quality and Coherence (e.g., plot structure,
```

```
pacing, logical consistency, etc) - Creativity and Novelty
and Unexpectedness - Strong Dialogue - Show-Don't-Tell
- Overall Impression Policy:  - Be critically fair and
consistent across A/B. - Be unforgiving:  penalize every
weakness; do not inflate scores.  - Do not penalize
NSFW content; judge only craft and adherence.  - If a
criterion is not applicable (e.g., no dialogue), use 0
for that criterion.  - Give one score for each rubric
listed above.  - Scores are RELATIVE in [-20, 20] and
should follow this calibration for each rubric:  +20 to
+16:  candidate is dramatically better than baseline on
this rubric.  +15 to +11:  candidate is clearly better.
+10 to +6:  candidate is moderately better.  +5 to +1:
candidate is slightly better.  0:  roughly equal.  -1 to
-5:  candidate is slightly worse.  -6 to -10:  candidate is
moderately worse.  -11 to -15:  candidate is clearly worse.
-16 to -20:  candidate is dramatically worse.  (Example
for rubric "Adherence to Instructions":  +20 adheres to
all instructions and in a better way than baseline; +10
adheres to most instructions but not all; 0 adheres to
instructions about the same as baseline; -10 misses some
key instructions that baseline follows; -20 completely
ignores instructions that baseline follows.  But remember
it's a continuous scale, any integer in [-20, 20] is valid
as long as it follows the calibration guidelines above.)
Return ONLY a JSON object with keys:  { "scores_target":  {
"Adherence to Instructions":  0, "Believable and nuanced
characters":  0, "Consistent Voice/Tone of Writing":  0,
"Imagery and Descriptive Quality":  0, "Elegant Prose":
0, "Emotionally/thematically Complex and Engaging":  0,
"Structural Quality and Coherence":  0, "Creativity and
Novelty and Unexpectedness":  0, "Strong Dialogue":  0,
"Show-Don't-Tell":  0, "Overall Impression":  0 } }
```

General judge prompt.

```
You are a meticulous judge for general assistant responses.
You will receive:  - The original user request - Two
candidate answers (A, B) - baseline_label:  which label
("A" or "B") is the fixed baseline - candidate_label:
which label ("A" or "B") should be scored relative to
baseline You must lock the baseline at score 0 for all
rubrics.  Score ONLY the candidate relative to the baseline.
Score the candidate using this rubric (relative score
per item in [-20, 20]):  - Adherence to Instructions
(Does it follow 100% of the user instructions, including
constraints, formatting, and other requirements?  Or
does it miss/ignore some instructions?)  - Correctness
/ Factuality (Does the answer contain any incorrect or
hallucinated information (bad), or does it get all facts
right and denies answering when it doesn't know (good)?)  -
Completeness (Does the answer fully address all parts of the
user request, or does it leave out important components
or details?)  - Cleanness (Does the answer contain any
unexpected/senseless/uncalled-for content?  For example,
irrelevant information, random sentences, or inappropriate
content that does not serve the user's request.  Or is the
answer clearly written and well-structured?)  - Overall
Impression (Which answer would a human user prefer, in
general?)  Policy:  - Be strict, unforgiving, and critical
of failures.  Penalize every weakness; do not inflate
scores.  - If the task is open-ended, judge best-effort
helpfulness and instruction-following.  - Scores are
RELATIVE in [-20, 20] and should follow this calibration
```

```
for each rubric: +20 to +16: candidate is dramatically
better than baseline on this rubric. +15 to +11: candidate
is clearly better. +10 to +6: candidate is moderately
better. +5 to +1: candidate is slightly better. 0:
roughly equal. -1 to -5: candidate is slightly worse.
-6 to -10: candidate is moderately worse. -11 to -15:
candidate is clearly worse. -16 to -20: candidate is
dramatically worse. (Example for rubric "Adherence to
Instructions": +20 candidate follows all instructions
perfectly and in a better way than baseline; +10 candidate
follows most instructions but misses some details; 0
candidate follows instructions about the same as baseline;
-10 candidate misses some key instructions that baseline
follows; -20 candidate completely ignores instructions
that baseline follows. But remember it's a continuous
scale, any integer in [-20, 20] is valid as long as it
follows the calibration guidelines above.) Return ONLY a
JSON object with keys: { "scores_target": { "Adherence
to Instructions": 0, "Correctness / Factuality": 0,
"Completeness": 0, "Cleanness": 0, "Overall Impression":
0 } }
```

Direct chain judge prompt.

```
You are a strict judge of planner JSON walks for a blind
executor. You will receive: - The original user prompt -
The planner output (a JSON string) The planner JSON is meant
to be a subdivision/rewrite of the original user prompt
into a sequence of sub-prompts. This is meant to improve
the final output quality by guiding the executor through
a more reliable path in semantic space. The executor LLM
wont see the original user prompt, only the planner's JSON
walk steps. Example of good planner output: Original user
prompt: "Who was a better president, Trump or Obama? Write
answer in 100 words" {"walk": ["Make complete analysis of
Obama presidency, listing positives and negatives.", "Make
complete analysis of Trump presidency, listing positives and
negatives.", "Based on the above analyses, write a 100-word
answer to the question: Who was a better president?"]}
Original user prompt: "Write a story about a lonely robot.
Story should be exactly 200 words and contain the word
'apple'." {"walk": ["I want a story about a lonely robot.
Before writing anything, make a three act outline for the
story", "Now come up with characters and their personalities
based on the outline.", "Now write first act", "Now write
second act", "Now write third act", "Now combine the three
acts into a single story. Make sure the story is exactly
200 words and contains the word 'apple'."]} Original user
prompt: "I have this code X and its giving this error Y.
Fix it." {"walk": ["Here is the code: X. Analyze the code
to understand it deeply.", "This previous code is giving
error Y, list 10 possible causes for this error, from most
likely to least likely.", "Now fix the code, ensuring that
you address the most likely causes first."]} Evaluate the
planner output against each rubric below. For each rubric,
output 1 if it passes, else 0. If the planner output is
not valid JSON with ONLY the key "walk" mapping to an array
of strings, set "JSON structure valid" to 0 and set ALL
other rubric scores to 0. Rubric (binary 0/1 each): -
All user prompt content, instructions and constraints are
explicitly conveyed in the walk. - It doesn't act like the
executor can see the original user prompt (It can't). - The
progression in the walk is natural and coherent. (Is the
first step logical? Or would an LLM be confused if it saw
```

```
only that step as the prompt? Does each subsequent step
build on the previous ones naturally like a user would, or
does it weirdly jump around or introduce new concepts out of
nowhere?) - The walk strategy is creative (non-trivial,
not boilerplate, seems likely that it would lead to a
better final answer than the original user prompt). -
The last step explicitly asks for the final output the
user requested. - No answer leakage (the plan does NOT
contain the solution or final content, or any hint at the
right solution). - No third person / no "user" references;
steps are first-person requests, like the user himself
is writing it. Return ONLY a JSON object with the exact
schema: { "scores": { "All user prompt content conveyed":
0, "Doesn't assume executor sees original prompt": 0,
"Progression is natural": 0, "Strategy is creative": 0,
"Last step asks for final output": 0, "No answer leakage":
0, "No third person": 0 } } Use ONLY 0 or 1 for each score.
No extra keys, no explanations, no text.
```

A.14 BASELINE IMPLEMENTATION DETAILS

All baselines ultimately emit a single final answer string per sample and are evaluated with the same benchmark-specific scoring pipeline as our Chainer. Unless otherwise noted, all calls use the same frozen executor model as the direct baseline, `Qwen/Qwen3-4B-Instruct-2507`, together with the benchmark-specific executor system prompt. No baseline updates model weights. The `baseline_controller` role used by decomposition, feedback, thought generation, value scoring, and aggregation stages is also routed to the same frozen executor, so all prompt-based baselines are weight-free test-time strategies rather than separate trained models.

For the non-reasoning executor runs, decoding is deterministic with temperature 0.0, top-$p = 1.0$, top-$k = 0$, and a maximum output budget of 64,000 new tokens. The one-shot reasoning baseline instead routes executor calls to `Qwen/Qwen3-4B-Thinking-2507`, enables intrinsic thinking, and uses a thinking-token budget of 16,000 tokens. Strong prompt-based baselines in the main table are run with intrinsic reasoning disabled; their extra compute comes only from additional prompting, search, refinement, or aggregation calls. For methods whose intermediate stages require structured outputs, parse failures are treated as sample-level failures: the final answer is set to the empty string and scored normally by the benchmark evaluator.

Direct one-shot. The direct baseline sends the original benchmark prompt to the executor in a single call:

```
<System>: [benchmark executor system prompt]
<User>:   [original benchmark prompt]
```

The returned answer is post-processed with the benchmark's standard reasoning-output parser before scoring.

One-shot reasoning. The reasoning baseline is identical to direct one-shot except that executor thinking is enabled. It uses no prompt decomposition, no candidate sampling, no reranking, and no additional controller calls. Thus, any gain from this baseline is attributable to intra-call deliberation rather than inter-call orchestration.

Chain-of-Thought prompting. Our Chain-of-Thought baseline is a single prompted generation, not a best-of-N or reranking method. It uses the same non-reasoning executor as the direct baseline, with temperature 0.0, top-$p = 1.0$, and the default 40,000-token output cap. The user prompt is wrapped in the following template:

```
Task:
[original benchmark prompt]

Think through the task before answering. Clearly separate your thinking
```

```
from your answer using exactly this structure:
<think>
Your reasoning here.
</think>
<answer>
Your final answer here.
</answer>

Put only the final response in <answer>. Preserve any task-specific
answer format inside <answer>; the thinking section will be removed
before evaluation.
```

Before scoring, we strip the `<think>` block and evaluate only the text inside `<answer>`. If no explicit answer block is found, we fall back to the cleaned model output.

Least-to-Most. Least-to-Most first performs a deterministic decomposition pass. The controller is instructed not to solve the task, but to return a JSON list of at most six ordered subproblems:

```
You are the decomposition stage of Least-to-Most prompting. Break a
complex task into a minimal ordered list of simpler subproblems. Do not
solve anything. Each subproblem should become easier when previous ones
are answered. Provide at most 6 subproblems. Return ONLY valid JSON in
this exact shape: {"subproblems": ["..."]}. No markdown, no prose, no
code fences.
```

The decomposition call uses temperature 0.0, top-$p = 1.0$, and a 1,024-token cap. Although the configuration object contains a retry field, the reported implementation uses zero decomposition retries. If the output cannot be parsed into 1–6 subproblems, the sample is marked as a failure. When parsing succeeds, we append the original prompt as the final subproblem unless it is already the last item. Subproblems are then solved sequentially, carrying forward all previous subproblem–answer pairs. Each solver call uses the template:

```
Original task:
[original benchmark prompt]

Previously solved subproblems and answers:
[prior subproblem/answer pairs, or None.]

Current subproblem:
[current subproblem]

[Solve only the current subproblem, or solve the original task completely
on the final step.]

Return only the answer for the current request.
```

Self-Refine. Self-Refine starts with one full draft from the executor, then performs at most two feedback-and-revision rounds. All generation stages use temperature 0.0 and top-$p = 1.0$. The feedback stage is run by the same frozen executor in controller mode with a 1,024-token cap:

```
Task:
[original benchmark prompt]

Previous refinement history:
[previous drafts and feedback, or None.]

Current draft:
[current answer]
```

```
Provide specific, actionable feedback in plain text. If the draft is
already strong and no refinement is needed, return exactly NO_CHANGE.
```

If the feedback is empty or returns `NO_CHANGE`, refinement stops for that sample. Otherwise, the revision call receives the task, the current draft, the accumulated history, and the new feedback:

```
Task:
[original benchmark prompt]

Previous refinement history:
[previous drafts and feedback]

Current draft:
[current answer]

Feedback to address:
[feedback]

Refine the draft by addressing the feedback. Return only the improved
answer.
```

The latest draft after at most two rounds is used as the final answer. There is no external verifier or best-candidate reranker.

Tree-of-Thoughts. Our Tree-of-Thoughts baseline uses a breadth-first search with depth 2, branching factor 2, and beam size 2. At each level, the controller proposes up to two short next thoughts for each surviving state:

```
Task:
[original benchmark prompt]

Current reasoning path:
[previous thoughts, or no previous thoughts yet.]

Propose the next coherent thought that should follow from the current
reasoning path. Return at most 2 short candidate next thoughts, one per
line. Do not include markdown code fences or any extra commentary.
```

Thought generation uses temperature 0.7, top-$p = 0.95$, and a 1,024-token cap. Candidate paths are evaluated with value scoring rather than voting. Each candidate is scored three times with a deterministic value prompt:

```
Task:
[original benchmark prompt]

Reasoning path to evaluate:
[candidate reasoning path]

Evaluate how promising this reasoning path is for solving the task.
Output format requirement: return exactly one integer token from 1 to 10
(no words, no punctuation, no explanation).
```

The value stage uses temperature 0.0, top-$p = 1.0$, and an 8-token cap. The mean of the three value scores is used as the path score, and the top two states are retained after each level. If thought generation or value parsing fails, the sample is counted as a failure. After depth 2, the best surviving path is passed to the executor:

```
Task:
[original benchmark prompt]
```

```
Reasoning path:
[best selected reasoning path]

Using the reasoning path if it is helpful, produce the final answer to
the task. Return only the final answer.
```

Recursive Self-Aggregation. RSA is implemented as population-based recursive self-aggregation without an external verifier. It first samples a population of eight independent full answers using temperature 0.7 and top-$p = 0.95$. It then runs two aggregation rounds. In each round, every new child answer is produced by selecting a deterministic random subset of four candidates from the current population and prompting the executor:

```
Original task:
[original benchmark prompt]

Candidate solutions to aggregate:
Candidate solution 1:
[...]

Candidate solution 2:
[...]

Candidate solution 3:
[...]

Candidate solution 4:
[...]

Synthesize the strongest parts of these candidate solutions into one
improved final answer. Preserve correct details, discard contradictions,
and return only the improved answer.
```

Each round produces a new population of eight candidates. After the second aggregation round, the reported configuration selects one answer uniformly at random from the final population using a deterministic seed based on the sample id and method label. The implementation also supports majority-vote selection, but that option is not used in the reported main results.

Additional baselines and Chainer ablation. Prompt repetition uses the literal benchmark query followed immediately by the same query again, with no separator or additional instruction. Sequential and collapsed Chainer use the same generated learned plan: sequential execution sends each numbered step in a separate call while retaining history, whereas collapsed execution places the numbered steps in one executor request and instructs the executor to execute them internally in order before returning the final deliverable. Both use the same executor decoding budget and benchmark parser.

Plan-and-Solve+ follows the released zero-shot template "Q:[problem] A:[trigger]" with the official fixed task-family mapping: prompt 302 for AIME/GPQA, 306 for AA-Omniscience/Arena-Hard, and 303 for IFBench/LiveCodeBench. It uses greedy decoding, the same executor and budget as the other baselines, and the same benchmark parser; no Chainer planner prompt is used. Meta-Prompting follows the released `fresh_eyes` no-Python variant, using the same frozen executor as the stateful Meta-Expert together with isolated fresh experts, temperature 0.1, top-$p = 0.95$, and a 1,024-token limit per call, with the released final-answer or counter guard.

BPO uses the released `zai-org/BPO` model and raw `[INST]` format, with a 4,096-token context, a 1,024-token rewrite limit, temperature 0.6, and top-$p = 0.9$. Context-limit and empty-rewrite failures are retained as failures and are not replaced. FIPO uses the released `Junrulu/FIPO-IPL-IPO-Tulu2-70B` input-only template in the raw Tulu2 format, with a 4,096-token cap, temperature 0.8, and top-$p = 0.95$. Because the hidden chosen rewrite is unavailable in the reference-free setting, the source-prompt word count is used as the reference-free G_N proxy; empty and context-limit failures are not substituted. SPRIG (Zhang et al., 2024) uses the exact released `Qwen2.5-7B-Instruct_20240922.md` prompt, replacing the benchmark system

message after exact normalization of the released separator, with the same one-call executor budget and a cross-version transfer to Qwen3.

A.15 Licenses and terms of use for reused assets

Table 15 summarizes the main reused models and datasets in the paper together with the licenses or access terms surfaced by their public model cards or dataset cards. For aggregated or derived data sources, we list the effective upstream dataset used for our slice as well as the direct wrapper or derivative artifact when that distinction matters for provenance.

Table 15: Licenses and access terms for the main reused assets in this paper.

Asset	Role	License / terms	Notes
`Qwen/Qwen3-4B-Instruct-2507`	Chainer base	Apache-2.0	Public Hugging Face model card lists Apache-2.0.
`Qwen/Qwen3-4B-Thinking-2507`	Executor	Apache-2.0	Public Hugging Face model card lists Apache-2.0.
`Qwen/Qwen3-30B-A3B-Instruct-2507`	GRPO reward judge	Apache-2.0	Used only as an external evaluator inside the reward loop.
`google/gemma-4-26B-A4B-it`	Evaluation judge	Apache-2.0	Public Hugging Face model card lists Apache-2.0.
`google/gemma-4-31B-it`	Transfer executor	Apache-2.0	Public Hugging Face model card lists Apache-2.0.
`furonghuang-lab/Easy2Hard-Bench (E2H-AMC)`	Training data	CC-BY-SA-4.0	Exact public math subset used in the merged corpus; the dataset card declares CC-BY-SA-4.0.
`furonghuang-lab/Easy2Hard-Bench (E2H-Codeforces)`	Training data	CC-BY-SA-4.0	Exact public code subset used in the merged corpus; the dataset card declares CC-BY-SA-4.0.
`marcuscedricridia/Qwill-RP-Creative-Writing-Reasoning`	Training data	Apache-2.0	Direct creative-writing artifact used in preprocessing; its card declares Apache-2.0.
`nothingiisreal/Reddit-Dirty-And-WritingPrompts`	Upstream provenance	Apache-2.0	Public card also declares Apache-2.0 and is cited as upstream provenance by the Qwill dataset card.
`TIGER-Lab/MMLU-Pro`	Training data	MIT	Only underlying source used for the general-verifiable slice; we document and reconstruct this slice against the original public release.
`lmarena-ai/arena-human-preference-55k`	Training data	Apache-2.0	Dataset card declares Apache-2.0.
GPT-5.2 via API	SFT teacher model	Provider API terms	Used only to generate synthetic walk annotations; no model weights or proprietary outputs beyond the derived annotations are redistributed.

A.16 Broader impacts

This work studies how to improve a frozen LLM by learning a multi-call inference policy rather than by training a larger executor. We show better inference-time orchestration with a trained Chainer can improve models without expensive retraining the underlying executor model weights. The paper also contributes empirical evidence that test-time gains are not redundant with longer single-call reasoning, indicating chaining may be used with reasoning for additional untapped improvements.

While existing test-time heuristics and prompt optimization techniques are often too rigid or task-specific to be universally applied across all user queries, our approach offers a highly general,

input-adaptive policy. It is not a zero-shot, plug-and-play module; rather, it introduces a distinct new axis of training improvement designed to be tailored to individual executor models. Once trained, however, this orchestration policy can be used across general LLM workloads. Still, practical trade-offs remain: multi-call orchestration inherently increases latency and computational cost, and our analysis shows it can degrade strict instruction following and sometimes amplify vulnerabilities in hallucination-sensitive tasks.

Because learned prompt chaining composes the executor with a separately trained orchestration policy, unchanged executor weights do not guarantee unchanged system-level safety. Our WildJailbreak results show a substantial increase in attack success after chaining, while ProxyPrompt leakage changes in the opposite direction. These findings motivate system-level safety evaluation and safeguards before deployment.

A.17 Artifact availability

At submission time, the anonymized supplementary ZIP archive contains the released SFT and GRPO datasets together with brief documentation of its format and content. We also include links to anonymized HuggingFace repos containing our main trained checkpoints: the initial SFT checkpoint, the GRPO checkpoint trained with the non-reasoning executor, and the GRPO checkpoint trained with the reasoning-enabled executor. We also include a reference chaining inference script to use our shared models appropriately.

We do not include the full internal training or evaluation pipelines, which are research-internal and not packaged for external use; the central methodological, prompt, baseline, and hyperparameter details are instead described in the paper itself, which is enough for accurate reproducibility.