LEAD: BREAKING THE NO-RECOVERY BOTTLENECK IN LONG-HORIZON REASONING

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon execution in Large Language Models (LLMs) remains unstable even when high-level strategies are provided. Evaluating on controlled algorithmic puzzles, we demonstrate that while decomposition is essential for stability, extreme decomposition creates a "no-recovery bottleneck". We show that this bottleneck becomes critical due to highly non-uniform error distribution, where consistent errors on a few "hard" steps become irreversible.

To address this, we propose Lookahead-Enhanced Atomic Decomposition (LEAD). By incorporating short-horizon future validation and aggregating overlapping rollouts, LEAD provides enough isolation to maintain stability while retaining enough local context to correct errors. This enables the `o4-mini` model to solve Checkers Jumping up to complexity $n = 13$, whereas extreme decomposition fails beyond $n = 11$.

1 INTRODUCTION

Large language models (LLMs) have demonstrated impressive performance on many reasoning benchmarks, yet their accuracy degrades rapidly on tasks that require executing long sequences of reasoning steps, even when each individual step is simple. This failure mode is especially pronounced in long-horizon algorithmic and puzzle-based tasks, where difficulty scales primarily with execution length rather than per-step complexity.

Prior works documented this phenomenon across diverse settings, including compositional factual queries (Press et al., 2023), mathematical reasoning (Hosseini et al., 2024; Zhou et al., 2025), and structured puzzle tasks (Shojaee et al., 2025). A consistent finding in recent literature is the significant discrepancy between the success probability of a composed task and the joint probability of its isolated subtasks—a phenomenon termed the *compositionality gap* (Press et al., 2023). Notably, this gap persists across different model scales (Press et al., 2023), suggesting that while scaling improves individual step accuracy, it does not inherently resolve the compounding error rates of long-horizon execution.

Recent works indicate that the failures in long-horizon reasoning often stem from execution errors rather than inability to plan. For example, Shojaee et al. (2025) show that providing an explicit solution strategy yields little improvement on algorithmic puzzle tasks, while Opus & Lawsen (2025) report that the very same models can generate a Python function that produces the complete solution to the puzzle, suggesting that high-level planning and algorithmic understanding remain intact. Taken together, these findings motivate a clear separation between *planning* and *execution*, and call for a focused investigation of execution reliability in isolation.

Reliable execution over long horizons is critical in many real-world applications, including program synthesis and refactoring, tool-using agents, and proof translation into formal language. In these settings, the high-level plan is often trivial or explicitly provided, yet failures arise from compounding errors in executing long sequences of simple, interdependent operations.

In this work, we aim to systematically understand the key principles behind robust long horizon execution. We start by distilling common techniques into two fundamental motifs: context truncation (periodically restarting the generation with only the essential summary required to proceed) and task decomposition. Specifically, we focus on Atomic Decomposition - an extreme form of decomposition where each step is executed in a separate model call using a minimal-context prompt.

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.

We evaluate these motifs on Checkers Jumping and Tower of Hanoi, two algorithmic puzzles with adjustable complexity and known optimal strategy, previously proposed as benchmarks by Shojaee et al. (2025). By providing the solution strategy in the prompt, we explicitly isolate execution from planning. These domains allow us to smoothly scale task complexity and analyze execution failures in a controlled setting, while serving as proxies for broader long-horizon reasoning challenges.

Our main contributions are summarized as follows:

- **The Necessity of Decomposition:** We demonstrate that for long-horizon execution, structural task decomposition is a prerequisite for stability. By comparing *Context Truncation* with *Atomic Decomposition*, we show that the deliberate isolation of reasoning steps—rather than mere context length management—is the primary driver of reliable execution across diverse models and benchmarks.
- **No-Recovery Bottleneck of Strict Decomposition:** While decomposition is necessary, we identify a fundamental limitation of extreme decomposition—where every step is executed in complete isolation: its memoryless design makes local errors irreversible. We show that this structural limitation is critical due to a highly non-uniform error distribution, where errors are concentrated on a few "hard" steps. Once a model becomes consistently wrong on even a single step, success becomes statistically impossible—a bottleneck that persists even with majority voting—despite the model remaining highly competent across the rest of the horizon.
- **Lookahead-Enhanced Atomic Decomposition (LEAD):** We propose LEAD, a framework that identifies the "Goldilocks zone" of task decomposition. By incorporating short-horizon validation and aggregating overlapping rollouts, LEAD provides enough isolation to maintain stability while retaining enough local context to correct errors. This enables the `o4-mini` model to solve Checkers Jumping up to complexity $n = 13$, whereas strict decomposition fails beyond $n = 11$.

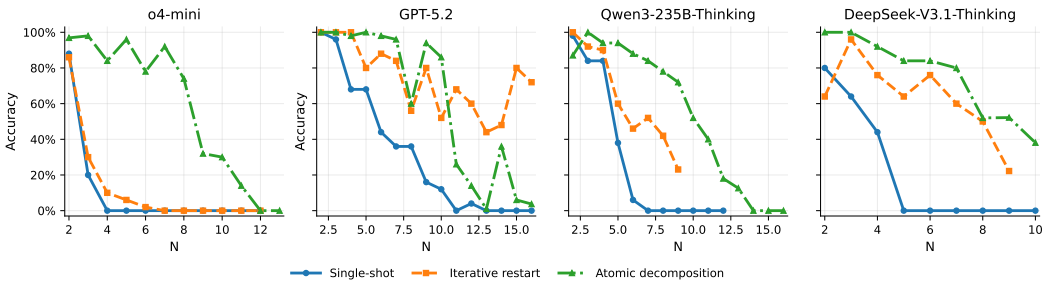
2 Related work

Long-Horizon Reasoning Failures in LLMs A growing body of work documents systematic performance degradation of LLMs on long-horizon tasks. One line of research attributes failures to degradation of model's reasoning under long context conditioning (Liu et al., 2024; Veseli et al., 2025; Du et al., 2025). Complementary work shows degradation under compositionality or multi-step reasoning setting. Press et al. (2023) introduce the *compositionality gap*, defined as the discrepancy between the success probability of a composed task and the product of success probabilities of its subtasks in isolation. Such gaps have been observed in two-hop factual composition (Press et al., 2023), mathematical reasoning (Hosseini et al., 2024; Zhou et al., 2025), and long-horizon puzzle tasks (Shojaee et al., 2025). Notably, the compositionality gap does not diminish with model scaling (Press et al., 2023), suggesting that reliable long-horizon execution does not emerge naturally through scale alone.

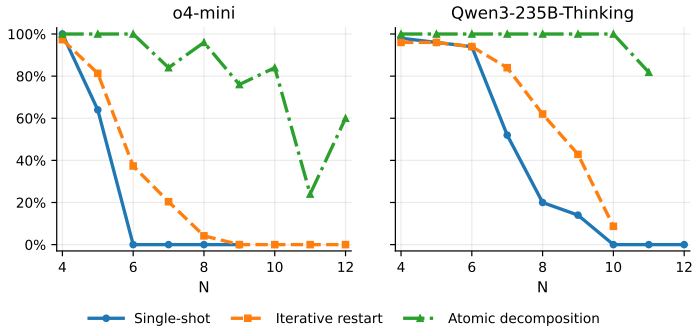
Context Management and Decomposition in LLM Reasoning Prior work addresses long-horizon reasoning through two complementary strategies: managing context growth and decomposing tasks into smaller components.

One line of research mitigates long-context degradation by explicitly reducing effective prompt length. Retrieve-then-reason frameworks first select relevant information and then perform reasoning over a shortened context (Li et al., 2024; Du et al., 2025). More generally, several agent architectures adopt bounded-context interaction loops, carrying forward only a compressed or structured representation of prior state to prevent unbounded context growth (Aghajohari et al., 2025; Liu et al., 2025).

A complementary line of work focuses on task decomposition. Prompting strategies such as Least-to-Most (Zhou et al., 2022), Plan-and-Solve (Wang et al., 2023), and Self-Ask (Press et al., 2023) decompose complex tasks into intermediate subproblems, typically executed within a single response that retains the full reasoning trace. Tree of Thoughts (Yao et al., 2023) extends this paradigm through tree-structured exploration using multiple model calls, emphasizing planning and search. Multi-agent systems further distribute subtasks across specialized agents and aggregate their outputs (Zhang et al., 2025; Wang et al., 2025; Meyerson et al., 2025).



(a) Checkers Jumping



(b) Tower of Hanoi

Figure 1: Comparison of methods across different problem domains

Solving Algorithmic Puzzles. Algorithmic puzzle domains have recently emerged as controlled testbeds for studying long-horizon reasoning and execution. Shojaee et al. (2025) introduce such puzzle benchmarks and demonstrate that model performance degrades sharply as problem complexity increases. Moreover, explicitly specifying solution plan in the prompt does not give any significant improvement. Sinha et al. (2025) argue that this degradation can be attributed to self-conditioning on earlier mistakes. Recent work by Meyerson et al. (2025) makes a major breakthrough in solving Tower of Hanoi puzzle using an extreme form of task decomposition with "micro-agents". Their framework is functionally equivalent to the *Atomic Decomposition*, which we will consider in Section 4, but augmented with a "first-to-ahead-by-k" voting scheme and a red-flagging mechanism for sample filtering. While their study focuses on maximizing task performance through extensive scaffolding, our work aims to isolate the fundamental execution bottlenecks that persist even under such decomposition.

3 TASKS

We use two algorithmic puzzles as our testbed: *Tower of Hanoi* and *Checkers Jumping*. **Tower of Hanoi** requires moving n disks between three pegs such that no larger disk is placed on a smaller one ($2^n - 1$ moves). **Checkers Jumping** involves swapping two blocks of n colored pieces on a 1D board via slides and jumps ($(n+1)^2 - 1$ moves). Full formal specifications and solution strategies are provided in Appendix H.

For both tasks, we provide the optimal strategy in the prompt and require the model to generate the full sequence of moves and resulting states without external tools. This structured format (see Listings 3, 4) allows us to distinguish between errors in move selection and state execution, enabling the fine-grained error analysis performed in Appendix B.

4 BASELINE METHODS

We study execution strategies for long-horizon reasoning under a fixed prompt prefix that specifies the puzzle description and its optimal solution strategy. Across all methods, only the task instance (e.g., problem size n and current state) varies.

Single-shot generation. In the single-shot setting, the model is prompted to generate the entire sequence of steps—including both moves and resulting states—in a single response. All intermediate reasoning and execution history remain within the same context window.

Iterative restart. In the iterative restart setting, the model is repeatedly prompted with the current state and asked to complete as many steps as it can. Within each response, the model may generate multiple consecutive steps until it either completes the task or reaches its capacity . In the latter case, the model outputs the partial solution, depending on how far it was able to proceed, and the updated state is fed back to the model in a new prompt. Iterative restart prevents unbounded context growth, but it does not guarantee the minimal-context execution since the model can solve multiple steps within a single response, thus accumulating the history of previous steps solution in reasoning traces.

Atomic Decomposition. Each reasoning step is executed in isolation: at step i, the model receives only state s_i to generate one move and next state s_{i+1}. By discarding prior history and intermediate traces, this approach eliminates context-based error accumulation and treats each step as an independent atomic operation.

Despite mitigating context degradation, Atomic Decomposition lacks a recovery mechanism: because steps are independent and history is discarded, errors at step i are irreversible. This structural limitation motivates augmenting atomic execution with short-horizon consistency checks that enable error correction without reintroducing large context dependencies.

5 LOOKAHEAD-ENHANCED ATOMIC DECOMPOSITION (LEAD)

To address the no-recovery bottleneck, we propose *Lookahead-Enhanced Atomic Decomposition (LEAD)*. This framework stabilizes execution by combining short-horizon future validation with consistency-filtering and aggregation across overlapping rollouts.

Lookahead mechanism. At step i, instead of predicting only the immediate next state, the model generates a rollout of k future steps: $(s_i \to s_{i+1} \to \cdots \to s_{i+k})$. By forecasting future states, the model can implicitly detect and revise inconsistencies; if a locally incorrect decision leads to future contradictions, the model may self-correct within the same rollout. This introduces forward validation while maintaining a bounded context length.

Overlapping aggregation. LEAD aggregates predictions across rollouts initiated from the current state i and the previous $h-1$ states. For each rollout, we enforce consistency with the committed trajectory by resampling until all predicted steps overlapping with the history match the executed steps. This *positive conditioning* (Section 6.3) ensures the model operates under the assumption of a correct prior path, significantly improving rollout accuracy.

Voting procedure. To optimize for computational efficiency, LEAD employs a two-tier verification process (Algorithm 1). For each step, the model first generates v independent atomic predictions for the current state s_i. If all samples reach a consensus, the model proceeds via a *fast-path*, accepting the step without further computation.

However, if disagreement is detected among the base samples—signaling a potential "hard step"—the framework escalates to the full lookahead mechanism. Predictions for the current state are then extracted from overlapping rollouts and aggregated via a margin-based voting procedure (Meyerson et al., 2025). Voting continues until one candidate outvotes all others by a margin t. This hierarchical structure ensures that the heavy computational cost of generating k-step rollouts is only incurred when necessary to resolve local instabilities. This process effectively smooths the error distribution: difficult steps that cause atomic disagreement are often correctly resolved when viewed through the wider context of rollouts originating from nearby states.

Algorithm 1 LEAD (Lookahead-Enhanced Atomic Decomposition)

Require: Model M, votes v, depth k, history h, margin t, initial state x
Ensure: Action sequence A

$A \leftarrow [\,]$
while not finished **do**
  ▷ Sample v independent next-step predictions for current state x
  $\{(a^{(m)}, x^{(m)})\}_{m=1}^{v} \sim M(\text{current state } x)$
  if all samples agree **then**
    $(a, x) \leftarrow (a^{(1)}, x^{(1)})$ ▷ Fast-path: accept consensus
  else
    $V[\cdot] \leftarrow 0$ ▷ Initialize vote counter
    while $\max V - \text{2nd-max } V < t$ **do**
      for x' in last h states **do**
        Sample a k-step rollout from x' (consistent with history)
        Let (a, x) be the step in this rollout corresponding to the current time
        $V[(a, x)] \mathrel{+}= 1$
      end for
    end while
    $(a, x) \leftarrow \arg\max V$ ▷ Resolve disagreement via lookahead consensus
  end if
  $A \leftarrow A \cup \{a\}$
end while
return A

6 EXPERIMENTS

Setup. We evaluate o4-mini, GPT-5.2 (both at low reasoning effort), Qwen3-235B-Thinking, and DeepSeek-V3.1-Thinking. Results are averaged over 50 independent runs, with the exception of DeepSeek-V3.1-Thinking, and Tower of Hanoi baselines (Atomic Decomposition and Iterative Restart), which use 25 runs due to computational cost. Detailed prompts are provided in Appendix H.

6.1 DECOMPOSITION IS NECESSARY FOR LONG-HORIZON STABILITY.

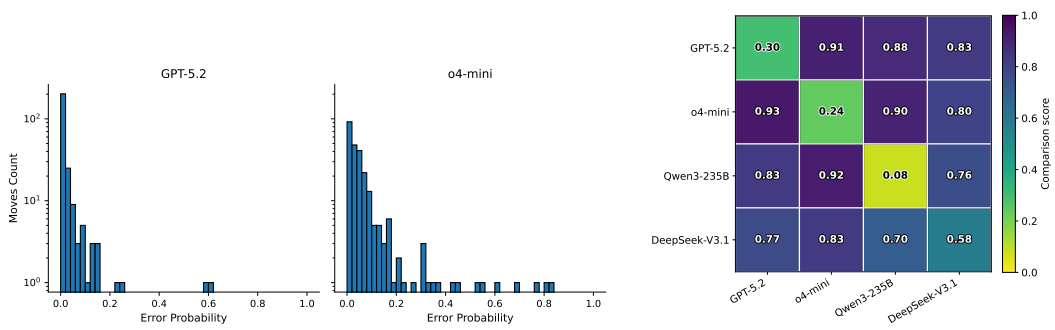
Figure 1 shows that Atomic Decomposition generally outperforms other baselines, confirming that isolating reasoning steps is essential for stability even under strict context truncation. A notable exception is GPT-5.2, where Iterative Restart performs better at higher complexities. We attribute this to the model's strong *positive-conditioning* effect (Section 6.3): by restarting at varying positions, the model can initiate generation from "easy" states. As shown in Figure 4b, this allows GPT-5.2 to solve subsequent hard steps that it otherwise fails to resolve when they are positioned at the start of a generation.

6.2 THE HARD-STEP BOTTLENECK

To understand why decomposition fails as complexity increases, we analyze per-step error distributions (see Appendices B, D for qualitative analysis). We find a fundamental divergence in difficulty between the two tasks. In **Tower of Hanoi**, error probabilities remain uniformly low (≤ 0.08 for `o4-mini` up to $n = 12$), explaining why stepwise execution with majority voting scale effectively (Meyerson et al., 2025).

Conversely, **Checkers Jumping** exhibits a highly non-uniform distribution. While most steps have near-zero error, a subset of "hard steps" emerges where error probabilities exceed 0.5 for `o4-mini` ($n \geq 12$) and `GPT-5.2` ($n = 15$), as shown in Fig. 2a. This concentration creates a *no-recovery bottleneck*: because errors are consistent rather than independent, naïve majority voting fails regardless of the number of votes, stalling the model at these critical junctions.

Model-specific heterogeneity. Our analysis suggests these hard steps are often model-specific rather than intrinsic to the task. We quantify this by calculating the pairwise Total Variation (TV)



(a) Within-model: error per step.

(b) Across-model: TV distance.

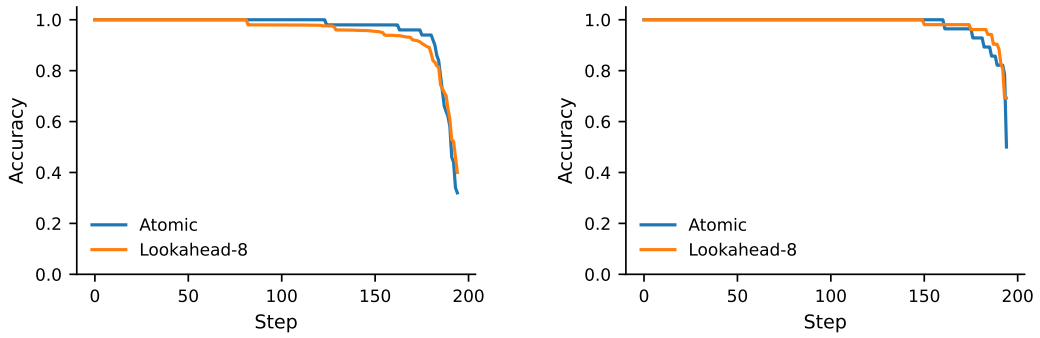
Figure 2: Non-uniform error distributions in Checkers Jumping. (a) Histogram for $o4$-mini and GPT-5.2 ($n = 15$) shows errors concentrated on specific "hard steps." (b) Pairwise Total Variation (TV) distance heatmap ($n = 13$) indicates that these hard steps vary significantly across models, while diagonal values confirm estimate robustness.

distance between empirical error distributions of different models on Checkers Jumping ($n = 13$). As shown in Fig. 2b, distributional divergence between different models significantly exceeds the internal variance of any single model. This indicates that different architectures struggle with **largely distinct** subsets of the state space. Such heterogeneity suggests that the "hard-step" barrier is not a universal limit of the task, but a behavioral signature of specific models, implying that ensembling or adaptive selection could further stabilize long-horizon execution.

6.3 Mechanisms of Lookahead Recovery

Lookahead (Section 5) functions as a targeted error-correction tool. As shown in Figure 3, while a $k = 8$ rollout may slightly degrade average-case accuracy for `o4-mini`, it consistently improves performance on the "hardest" steps for both `o4-mini` and `GPT-5.2`. This confirms our intuition: while forecasting adds complexity to trivial steps, it provides a critical corrective signal at high-entropy junctions where atomic intuition fails (see App. E).

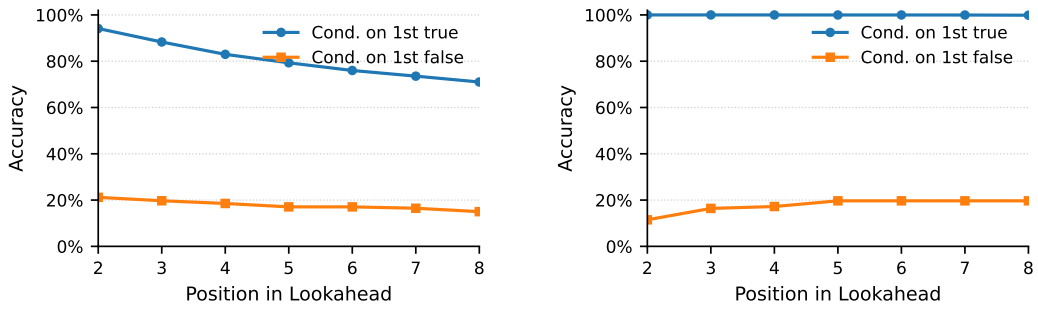
The efficacy of this signal is governed by two competing phenomena. First, **positional accuracy decay**: for `o4-mini`, accuracy typically degrades as the prediction moves further from the current state (Figure 4a). Second, a powerful **self-conditioning effect** can counteract this decay. For `GPT-5.2`, if the initial rollout step is correct, the model "locks onto" a valid trajectory, maintaining high stability across the horizon (Figure 4b). This effect is also present in `o4-mini`, albeit more



(a) o4-mini

(b) GPT-5.2

Figure 3: Rank-ordered accuracy plots of Atomic decomposition and Lookahead ($k = 8$) on Checkers Jumping ($n = 13$). While Lookahead slightly degrades the average accuracy for o4-mini models, it **boosts the performance on the hardest moves** for both models.



(a) o4-mini (b) GPT-5.2

Figure 4: Per-position accuracy of Lookahead predictions for Checkers Jumping ($n = 13$), conditioned on whether the first position is correct ("Cond. on 1st true") or incorrect ("Cond. on 1st false").

		o4-mini (N)			*GPT-5.2* (N)				
Method	Voting	11	12	13	13	14	15	16	17
AD	×	14	0	0	0	36	0	4	0
AD	√	76	16	0	20	**100**	0	96	0
LEAD	√	**100**	**56**	**76**	**100**	**100**	**100**	**100**	**80**

Table 1: Performance (%) of Atomic Decomposition (AD) and Lookahead-Enhanced Atomic Decomposition (LEAD) on Checkers Jumping with first-to-ahead-by-3 voting. LEAD uses $k = 8$; $h = 2$, $t = 3$ and $v = N$ for both models.

weakly. LEAD exploits this by enforcing consistency with committed steps, ensuring rollouts benefit from this positive conditioning.

6.4 Main results: surpassing the hard-step-bottleneck

Table 1 shows that LEAD consistently outperforms both vanilla Atomic Decomposition and its majority-voting variant across the critical complexity thresholds where standard methods collapse. By neutralizing the "hard-step" failures that cripple decomposition, LEAD extends the reliable execution horizon and achieves a new state-of-the-art on the Checkers Jumping benchmark. These results demonstrate that while minimal context is often prioritized for stability, a hybrid approach—selectively incorporating the self-corrective signal of lookahead rollouts (Section 5)—is essential to navigate irreversible failure points in long-horizon reasoning.

For these experiments, we configured LEAD with parameters $h = 2$, $k = 8$, and $t = 3$, setting the base votes v equal to the task complexity n. Ablations using a larger history window ($h = 3$) yielded no significant performance gains (see Table 2). To ensure a rigorous comparison, the Atomic Decomposition baseline was equipped with a competitive "first-to-ahead-by-3" voting scheme (Meyerson et al., 2025). Despite this, LEAD's cross-step aggregation provided a superior error-correction signal, effectively "rescuing" trajectories that otherwise hit the $n = 12$ bottleneck when using `o4-mini`.

A notable exception occurred during the evaluation of GPT-5.2 at $n = 17$, where maintaining the $v = n$ strategy resulted in near-zero accuracy due to frequent false consensus during the base voting procedure (see Figure 8). To address this while keeping the experiment computationally tractable, we modified the sampling strategy by pre-generating a pool of 70 independent predictions for each step. During LEAD evaluation, we then sampled $v = 70$ candidates with replacement from this fixed pool for the base voting stage. This configuration significantly reduced the probability of an incorrect "fast-path" agreement, effectively shifting the primary decision-making responsibility to the more robust lookahead-based voting phase.

6.5 Inference cost analysis

Figure 5 illustrates the average token count per solution for the methods in Table 1. While LEAD is computationally more expensive than the baselines, it provides a unique capability: extending the reliable execution horizon beyond the ”hard-step” threshold.

As established in Section 6.2, scaling via simple stepwise voting fails once a model's error distribution becomes consistent at critical junctions. Any decomposition that executes steps in total isolation eventually hits this ”no-recovery” wall. To our knowledge, LEAD is the first algorithm to surpass this threshold by replacing isolated steps with overlapping rollouts. While we prioritized performance over inference efficiency, the per-solution token estimates (Appendix F) provide a baseline for future research into more efficient paradigms for scaling test-time compute in long-horizon tasks.

6.6 Ablation Studies

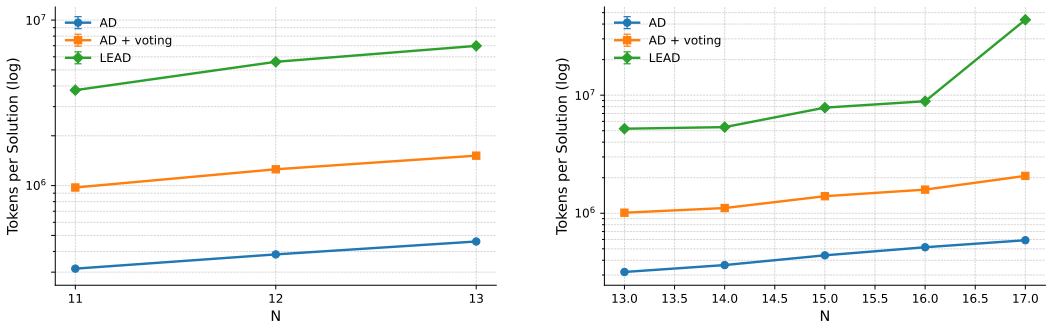
To evaluate the individual contributions of LEAD's core components—*lookahead*, *consistency filtering*, and *overlapping aggregation*—we compare the full framework against ablated variants across the critical complexity ranges where standard Atomic Decomposition typically fails. In the “no-lookahead” baseline, rollouts are truncated to the minimum length required to aggregate predictions for the current step, effectively isolating the impact of future-state forecasting.

As shown in Table 2, optimal performance is achieved only when all three mechanisms are integrated, validating our architectural design. Our results indicate that while the filtering component provides incremental improvements, overlapping aggregation is the primary driver of robust performance, as it enables the cross-step error correction necessary to navigate high-complexity trajectories.

7 Conclusion

In this work, we identified the no-recovery bottleneck as a primary obstacle to Atomic Decomposition in long-horizon execution. Our empirical analysis reveals that success is not limited by average model competence, but by a highly non-uniform error distribution where a few “hard” steps act as irreversible points of failure. We demonstrated that while minimal-context atomic execution provides stability for uniform tasks like the Tower of Hanoi, it lacks the necessary corrective signal to navigate the high-entropy junctions found in more complex puzzles.

To address this, we introduced LEAD (Lookahead-Enhanced Atomic Decomposition). By integrating temporal rollouts with consistency filtering and smoothed voting mechanism, LEAD identifies and corrects local errors before they propagate into global failures. Our results show that LEAD significantly extends the reliable reasoning horizon of frontier models like `GPT-5.2` and `o4-mini`, outperforming existing stepwise baselines. Ultimately, our findings suggest that the next frontier in



(a) o4-mini (b) GPT-5.2

Figure 5: Average number of tokens per solution for Atomic Decomposition (AD), Atomic Decomposition with first-to-ahead-by-3 voting (AD + voting), and LEAD (using the same parameters as in Table 1). Token counts are shown on a logarithmic scale.

Lookahead	Filtering	Aggregation	o4-mini (N)			GPT-5.2 (N)		
			11	12	13	13	14	15
			$h = 3$					
✓	✓	✓	96	**56**	**84**	100	100	**100**
✗	✓	✓	**100**	36	24	100	100	**100**
✓	✗	✓	96	52	80	100	100	92
✓	✓	✗	96	4	0	72	92	60
			$h = 2$					
✓	✓	✓	100	56	**76**	100	100	**100**
✗	✓	✓	100	32	28	100	100	32
✓	✗	✓	100	**64**	52	100	100	80
✓	✓	✗	96	4	0	72	92	60

Table 2: Performance comparison (%) of ablations for o4-mini and GPT-5.2. We used parameters $k = 8,\ t = 3,\ v = N$ for all runs, and varied aggregation window h in $\{2, 3\}$. Best results for each model within each h block are shown in bold. For "no-aggregation" ablation, we reported the result of the same simulation for both h-blocks, since this ablation is functionally independent of h.

robust AI execution lies not in further context reduction, but in adaptive motifs that can selectively leverage lookahead to stabilize critical transitions.

REFERENCES

Milad Aghajohari, Kamran Chitsaz, Amirhossein Kazemnejad, Sarath Chandar, Alessandro Sordoni, Aaron Courville, and Siva Reddy. The markovian thinker. In *The Fourteenth International Conference on Learning Representations*, 2025.

Yufeng Du, Minyang Tian, Srikanth Ronanki, Subendhu Rongali, Sravan Bodapati, Aram Galstyan, Azton Wells, Roy Schwartz, Eliu A Huerta, and Hao Peng. Context length alone hurts llm performance despite perfect retrieval. *arXiv preprint arXiv:2510.05381*, 2025.

Arian Hosseini, Alessandro Sordoni, Daniel Toyama, Aaron Courville, and Rishabh Agarwal. Not all llm reasoners are created equal. *arXiv preprint arXiv:2410.01748*, 2024.

Huayang Li, Pat Verga, Priyanka Sen, Bowen Yang, Vijay Viswanathan, Patrick Lewis, Taro Watanabe, and Yixuan Su. Alr^2: A retrieve-then-reason framework for long-context question answering. *arXiv preprint arXiv:2410.03227*, 2024.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.

Shukai Liu, Jian Yang, Bo Jiang, Yizhi Li, Jinyang Guo, Xianglong Liu, and Bryan Dai. Context as a tool: Context management for long-horizon swe-agents. *arXiv preprint arXiv:2512.22087*, 2025.

Elliot Meyerson, Giuseppe Paolo, Roberto Dailey, Hormoz Shahrzad, Olivier Francon, Conor F Hayes, Xin Qiu, Babak Hodjat, and Risto Miikkulainen. Solving a million-step llm task with zero errors. *arXiv preprint arXiv:2511.09030*, 2025.

C Opus and A Lawsen. The illusion of the illusion of thinking. *arXiv preprint ArXiv:2506.09250*, 2025.

Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah A Smith, and Mike Lewis. Measuring and narrowing the compositionality gap in language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 5687–5711, 2023.

Parshin Shojaee, Iman Mirzadeh, Keivan Alizadeh, Maxwell Horton, Samy Bengio, and Mehrdad Farajtabar. The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity. *arXiv preprint arXiv:2506.06941*, 2025.

Akshit Sinha, Arvindh Arun, Shashwat Goel, Steffen Staab, and Jonas Geiping. The illusion of diminishing returns: Measuring long horizon execution in LLMs. *arXiv preprint arXiv:2509.09677*, 2025.

Blerta Veseli, Julian Chibane, Mariya Toneva, and Alexander Koller. Positional biases shift as inputs approach context window limits. *arXiv preprint arXiv:2508.07479*, 2025.

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. *arXiv preprint arXiv:2305.04091*, 2023.

Qian Wang, Tianyu Wang, Zhenheng Tang, Qinbin Li, Nuo Chen, Jingsheng Liang, and Bingsheng He. Megaagent: A large-scale autonomous llm-based multi-agent system without predefined sops. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 4998–5036, 2025.

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, 2023. *URL https://arxiv. org/abs/2305.10601*, 3:1, 2023.

Wentao Zhang, Ce Cui, Yilei Zhao, Rui Hu, Yang Liu, Yahui Zhou, and Bo An. Agentorchestra: A hierarchical multi-agent framework for general-purpose task solving. *arXiv e-prints*, pp. arXiv–2506, 2025.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022.

Hattie Zhou, Arwen Bradley, Etai Littwin, Noam Razin, Omid Saremi, Josh Susskind, Samy Bengio, and Preetum Nakkiran. What algorithms can transformers learn? a study in length generalization. *arXiv preprint arXiv:2310.16028*, 2023.

Yang Zhou, Hongyi Liu, Zhuoming Chen, Yuandong Tian, and Beidi Chen. Gsm-infinite: How do your llms behave over infinitely increasing context length and reasoning complexity? *arXiv preprint arXiv:2502.05252*, 2025.

A FULL ALGORITHM DESCRIPTION

To formally define the LEAD framework, we introduce two prompt construction functions, ϕ_{AD} and ϕ_{LA}. The function $\phi_{\mathrm{AD}}(x)$ maps a single state x to a prompt following the Atomic Decomposition (AD) paradigm, instructing the model to predict exactly one next step (a, x'). The function $\phi_{\mathrm{LA}}(x, k)$ maps a state x and a lookahead depth k to a prompt instructing the model to generate a sequence of k consecutive steps starting from x.

Algorithm 2 LEAD (Lookahead-Enhanced Atomic Decomposition)

Require: Model M, vote count v, lookahead depth k, history window $h \leq k$, margin t, initial state x_0, horizon S
Ensure: Action sequence A, state sequence X

$A \leftarrow [\perp], \quad X \leftarrow [x_0]$ ▷ $\perp$ denotes a null action used only to align indices
$x \leftarrow x_0$
for $i = 1$ to S **do**
  $\{y^{(m)}\}_{m=1}^{v} \sim M\big(\phi_{\mathrm{AD}}(x)\big)$ ▷ $y^{(m)} = (a^{(m)}, x^{(m)})$
  if $y^{(1)} = \cdots = y^{(v)}$ **then**
    $(a, x) \leftarrow y^{(1)}$ ▷ If atomic samples agree, execute directly
  else
    $(a, x) \leftarrow \textsc{LookaheadVote}(M, A, X, h, k, t)$ ▷ Resolve disagreement using lookahead aggregation
  end if
  Append a to A
  Append x to X
end for
return A, X

Procedure 3 LOOKAHEADVOTE

Require: Model M, history (A, X), history window h, lookahead depth k, margin t
Ensure: Selected pair $y^\star = (a^\star, x^\star)$ for current step

$V[y] \leftarrow 0$ for all candidate pairs y
$i \leftarrow |X| - 1$ ▷ Current state index; next decision is for step $i + 1$
while True **do**
  for $\ell = 0$ to $\min(h - 1, i)$ **do**
    $y^{(\ell)} \leftarrow \textsc{ConsistentRollout}(M, A, X, i - \ell, k)$ ▷ Candidate induced by rollout started at state $x_{i-\ell}$
    $V[y^{(\ell)}] \leftarrow V[y^{(\ell)}] + 1$
  end for
  $y^\star \leftarrow \arg\max_y V[y]$
  if $V[y^\star] \geq t + \max_{y' \neq y^\star} V[y']$ **then**
    return $y^\star$ ▷ Return once one candidate leads by margin t
  end if
end while

Procedure 4 CONSISTENTROLLOUT

Require: Model M, history (A, X), rollout start index i_0, lookahead depth k
Ensure: Candidate pair for the next undecided step

$r \leftarrow (|X| - 1) - i_0$ ▷ Number of already-committed steps after rollout start
Define $a_j \leftarrow A[j]$, $x_j \leftarrow X[j]$ for all $j = 0, \ldots, |X| - 1$
while True **do**
  $(\hat{y}_{i_0+1}, \ldots, \hat{y}_{i_0+k}) \sim M\big(\phi_{\mathrm{LA}}(x_{i_0}, k)\big)$ ▷ Sample a k-step lookahead rollout
  if $\hat{y}_{i_0+j} = (a_{i_0+j}, x_{i_0+j})$ for all $j = 1, \ldots, r$ **then**
    return $\hat{y}_{i_0+r+1}$ ▷ Return the first prediction beyond the committed prefix
  end if
end while

B DETAILS ON ERROR ANALYSIS.

We distinguish two sources of failure: *move finding* (selecting the correct action) and *move execution* (correctly updating the state based on that action). We found that for Checkers Jumping the errors are dominated by move execution failures, whereas for Tower of Hanoi errors arise primarily from incorrect move selection (see Figure 10 in Appendix B).

Manual inspection suggests that execution errors in Checkers Jumping primarily arise from incorrect updates to long blocks of same-color checkers (see Appendix D for representative examples). This behavior aligns with known limitations of Transformer models in copying long sequences of identical symbols Zhou et al. (2023). In contrast, the Tower of Hanoi representation assigns a unique numerical identifier to each disk, which makes state updates substantially easier for the model. We hypothesize that introducing unique identifiers for checkers (e.g., $R_1, \ldots, R_N, B_1, \ldots, B_N$) or explicitly enumerating board positions (e.g., $[1{:}R, 2{:}R, \ldots, 2n{:}B]$) could mitigate these errors. We leave a systematic investigation of these alternatives to future work.

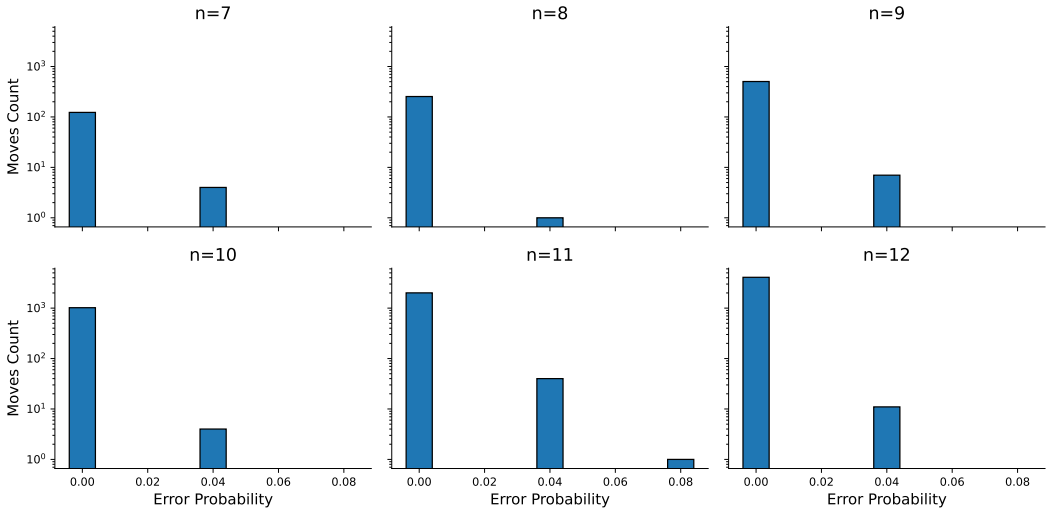


Figure 6: Error count histogram of o4-mini model for Tower of Hanoi. The error counts were estimated by sampling 25 independent solutions for each step.

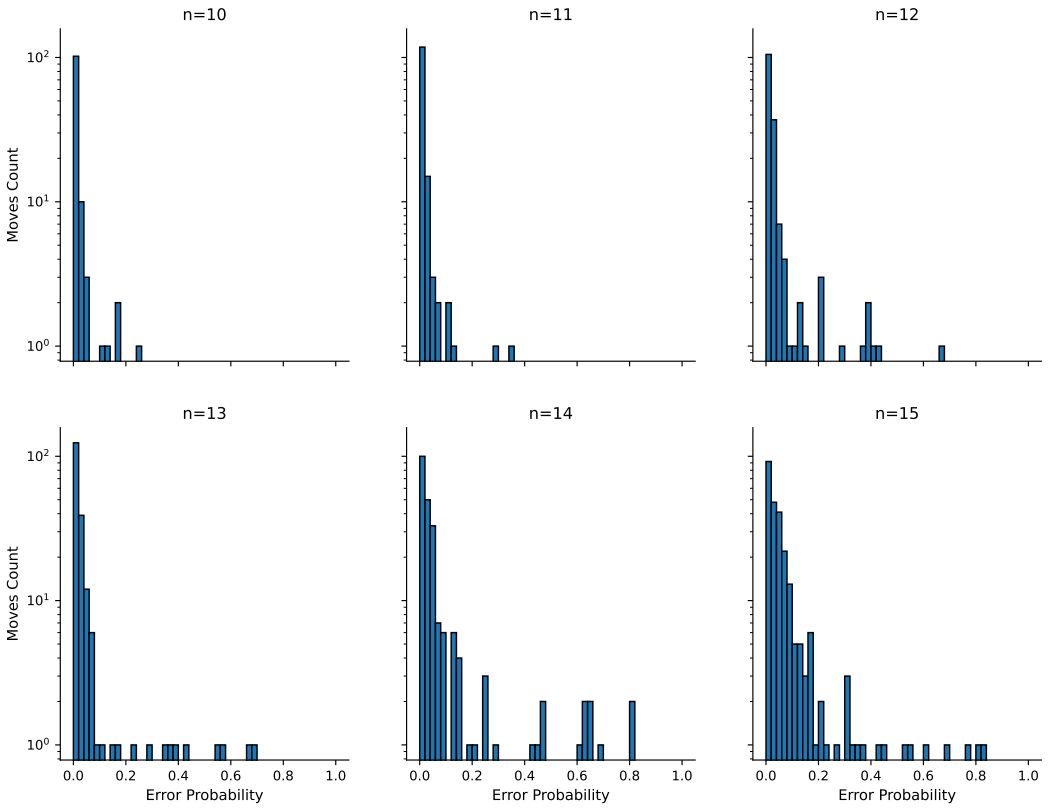


Figure 7: Error count histogram of o4-mini model for Checkers Jumping. The error counts were estimated by sampling 50 independent solutions for each step.

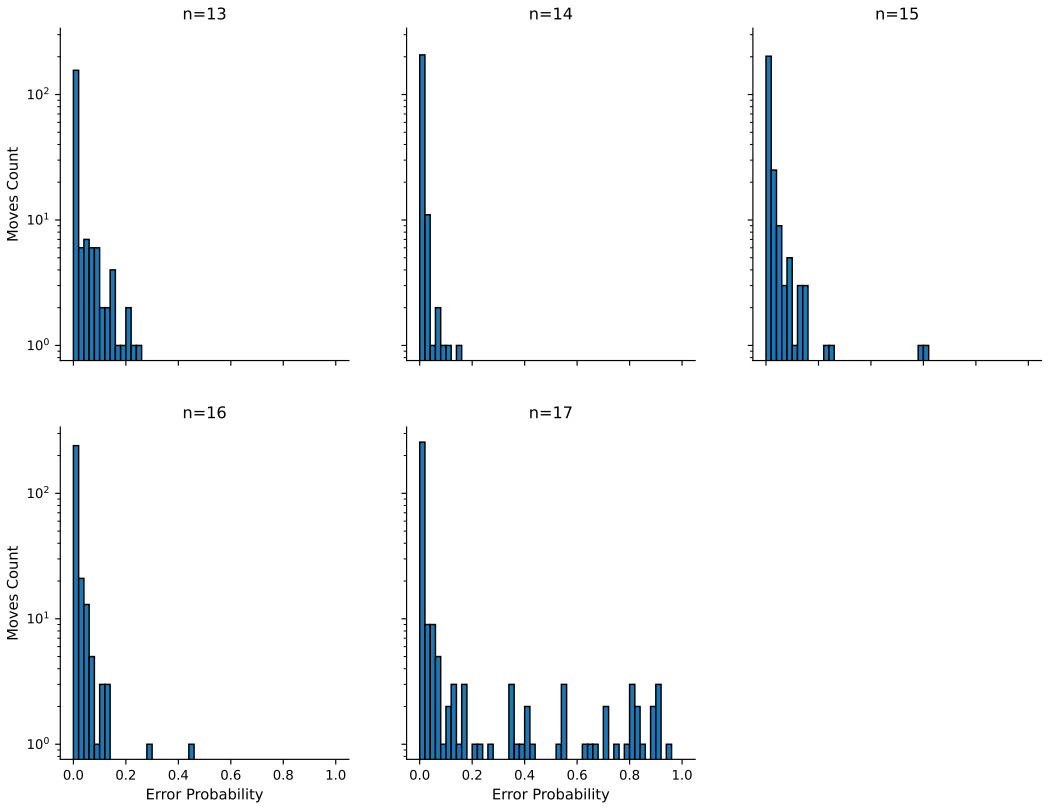


Figure 8: Error count histogram of GPT-5.2 model for Checkers Jumping. The error counts were estimated by sampling 50 independent solutions for each step.

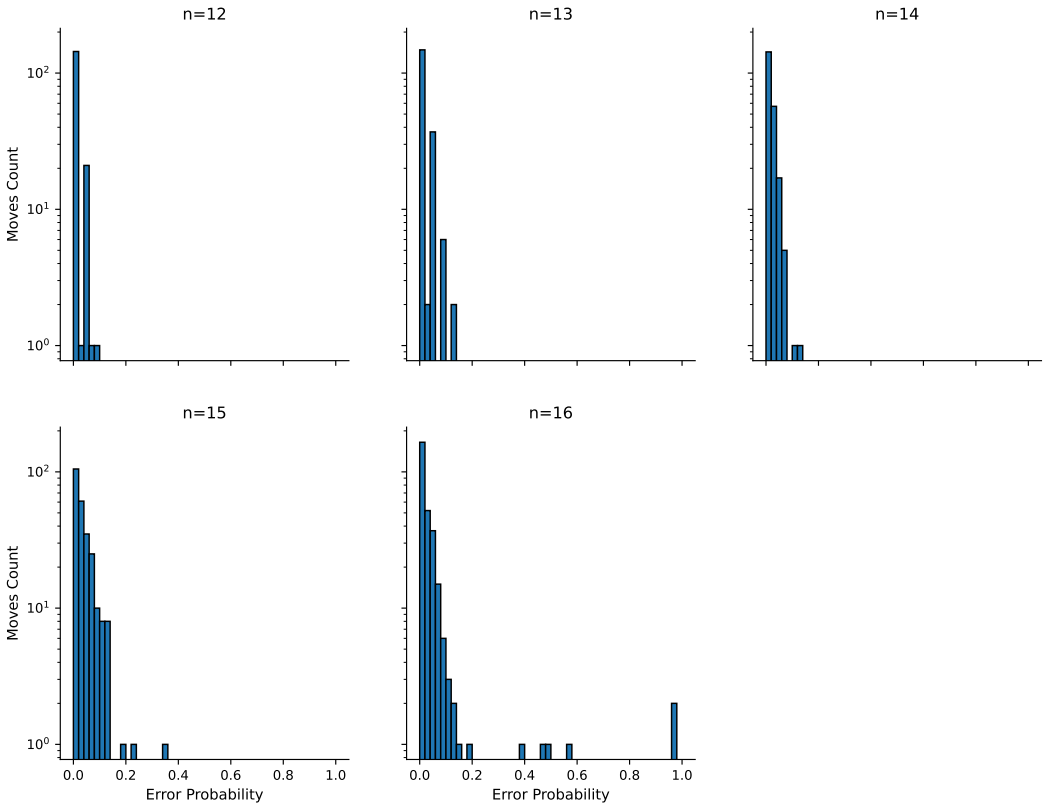
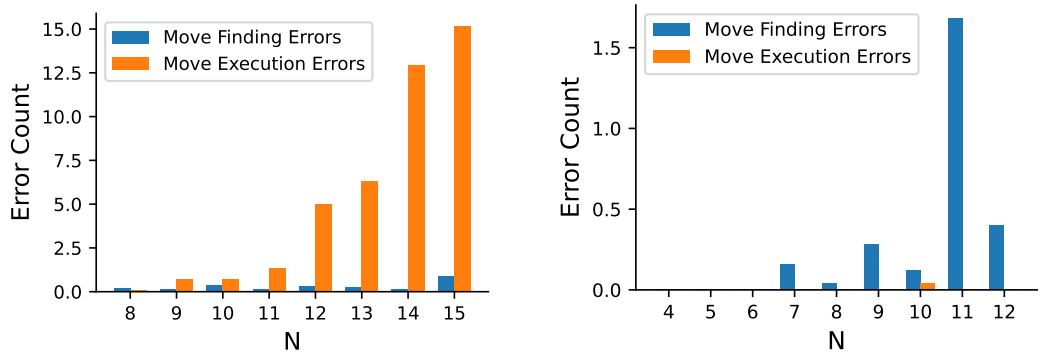


Figure 9: Error count histogram of Qwen model for Checkers Jumping. The error counts were estimated by sampling 50 independent solutions for each step.



(a) Checkers Jumping

(b) Tower of Hanoi

Figure 10: Distribution of error types for o4-mini model. Each plot shows the average number of each type errors per one solution. The main source of errors for Checkers Jumping is move execution, while for Tower of Hanoi - finding the correct move.

C ATOMIC COMPETENCE BARRIER

When Atomic Decomposition is combined with majority voting at each step, its failure mode converges to the dominant source of error. As shown in Figure 10, execution errors—rather than incorrect move selection—are the primary failure mode on Checkers Jumping. This suggests that the performance of Atomic Decomposition with voting is fundamentally bottlenecked by the model's ability to reliably execute a given move.

To substantiate this claim, we evaluate Atomic Decomposition with stepwise voting over $v = 32$ votes under three settings: (i) full-step execution (move finding followed by move execution), (ii) move finding only, and (iii) move execution only. Figure 11 shows that full-step execution closely approaches the performance of move execution alone, confirming that move execution constitutes the limiting factor. We refer to this limitation as the *atomic competence barrier*.

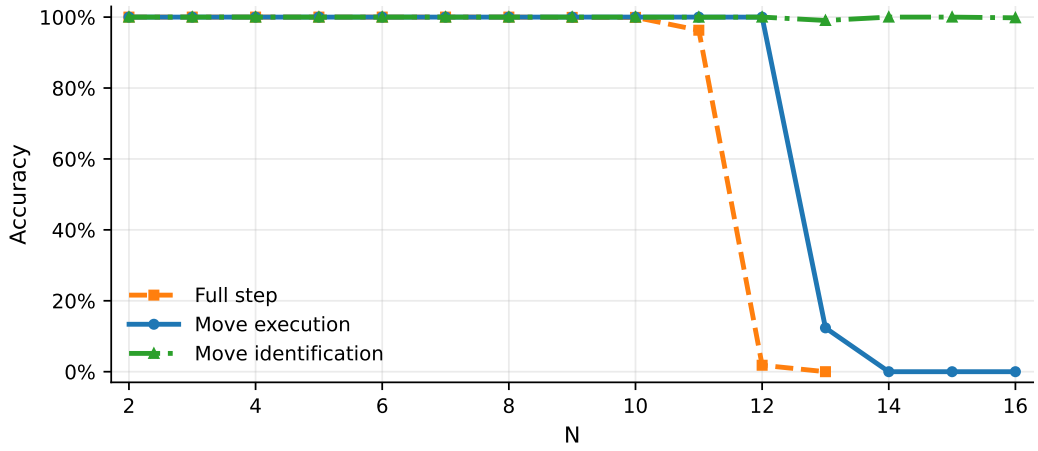


Figure 11: Atomic Decomposition with voting approaches the atomic competence barrier on Checkers Jumping, where the limiting factor is atomic move execution. The plot compares accuracy of full-step execution (selecting and executing a move) and isolated move selection or move execution as a function of problem size n. Results are shown for the o4-mini model using majority voting over 32 independently sampled solutions at each step.

D EXAMPLE OF ERRORS

Listing 1: The outputs of o4-mini model for two of the most difficult steps for this model for Checkers Jumping ($n = 13$), sampled 50 times per move. The main cause of error is forgetting one red checker 'R' in a long block of consecutive red checkers, when one more red checker should join the block as a result of the move. Thus, the model outputs the board of shorter length than it should be.

```
Step id = 168

Correct solution:
move = ['R', 17, 19]
state = ['B', 'B', 'B', 'B', 'B', 'B', 'B', 'R', 'B', 'R', 'B', 'R', 'B',
     'R', 'B', 'R', 'B', '_', 'B', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'R
    ']

Model solutions:
move = ['R', 17, 19]
state = ['B', 'B', 'B', 'B', 'B', 'B', 'B', 'R', 'B', 'R', 'B', 'R', 'B',
     'R', 'B', 'R', 'B', '_', 'B', 'R', 'R', 'R', 'R', 'R', 'R', 'R']
Count: 34

move = ['R', 17, 19]
state = ['B', 'B', 'B', 'B', 'B', 'B', 'B', 'R', 'B', 'R', 'B', 'R', 'B',
     'R', 'B', 'R', 'B', '_', 'B', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'R
    ']
Count: 16

Step id = 150

Correct solution:
move = ['R', 21, 22],
state = ['B', 'B', 'B', 'B', 'B', 'R', 'B', 'R', 'B', 'R', 'B', 'R', 'B',
     'R', 'B', 'R', 'B', 'R', 'B', 'R', 'B', '_', 'R', 'R', 'R', 'R', 'R
    ']
Model solutions:

move = ['R', 21, 22],
state = ['B', 'B', 'B', 'B', 'B', 'R', 'B', 'R', 'B', 'R', 'B', 'R', 'B',
     'R', 'B', 'R', 'B', 'R', 'B', 'R', 'B', '_', 'R', 'R', 'R', 'R']
Count: 33

move = ['R', 21, 22],
state = ['B', 'B', 'B', 'B', 'B', 'R', 'B', 'R', 'B', 'R', 'B', 'R', 'B',
     'R', 'B', 'R', 'B', 'R', 'B', 'R', 'B', '_', 'R', 'R', 'R', 'R', 'R
    ']
Count: 17
```

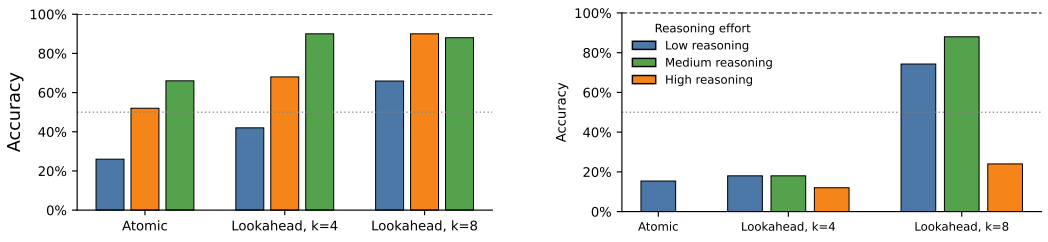
Listing 2: The outputs of GPT-5.2 model for two the most difficult steps for this model for Checkers Jumping ($n = 13$), sampled 100 times per move. The main cause of error is adding one extra blue checker 'B' in a long block of consequtive blue checkers. Thus, the model outputs the board of bigger length than it should be.

```
Step id = 16

Correct solution:
move = ['B', 12, 10],
state = ['R', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'B', 'R', 'B', 'R', '_',
     'R', 'B', 'R', 'B', 'R', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B
    ']
Model solutions:

move = ['B', 12, 10],
state = ['R', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'B', 'R', 'B', 'R', '_',
     'R', 'B', 'R', 'B', 'R', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B
    ']
Count: 78

move = ['B', 12, 10],
state = ['R', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'B', 'R', 'B', 'R', '_',
     'R', 'B', 'R', 'B', 'R', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B
    ', 'B']
Count: 22

Step id = 17

Correct solution:
move = ['B', 14, 12],
state = ['R', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'B', 'R', 'B', 'R', 'B',
     'R', '_', 'R', 'B', 'R', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B
    ']
Model solutions:

move = ['B', 14, 12],
state = ['R', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'B', 'R', 'B', 'R', 'B',
     'R', '_', 'R', 'B', 'R', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B
    ']
Count: 76

move = ['B', 14, 12],
state = ['R', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'B', 'R', 'B', 'R', 'B',
     'R', '_', 'R', 'B', 'R', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B
    ', 'B']
Count: 22
```

E MORE EXPERIMENTS WITH LOOKAHEAD

	Lookahead Accuracy by Position							
STEP IDX	+1	+2	+3	+4	+5	+6	+7	+8
60	1.0	1.0	1.0	**1.0**	1.0	1.0	1.0	1.0
61	1.0	1.0	**1.0**	1.0	1.0	1.0	1.0	1.0
62	0.74	**0.74**	0.74	0.74	0.74	0.74	0.73	0.73
63	**0.5**	0.5	0.5	0.5	0.5	0.5	0.5	0.5
64	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94

Table 3: Lookahead prediction accuracy for GPT-5.2 on Checkers Jumping ($n = 15$), shown for the hardest step (step index 63) and its neighboring steps. Columns correspond to the position within the Lookahead rollout. Bold entries indicate predictions for the hardest step obtained from different Lookahead starting points. Although the hardest step is not consistently correct at its native position, it is predicted correctly when inferred from earlier Lookahead rollouts, motivating the inclusion of recent Lookahead predictions in the voting procedure used by LEAD.



(a) o4-mini (b) GPT-5.2

Figure 12: Comparing Lookahead with Atomic Decomposition strategies for two models on the most difficult step for that model for Checkers Jumping puzzle (the puzzle complexity is $n = 13$ for o4-mini and $n = 15$ for GPT-5.2 model).

F INFERENCE COST ANALYSIS

	Complexity n		
Method	11	12	13
AD	3.15 $\pm$ 0.04	3.84 $\pm$ 0.05	4.60 $\pm$ 0.05
AD + voting	9.70 $\pm$ 0.19	12.70 $\pm$ 0.34	14.98 $\pm$ 0.31
LEAD	37.73 $\pm$ 0.99	55.96 $\pm$ 1.44	69.84 $\pm$ 1.42

Table 4: Estimated tokens per solution ($\times 10^5$) for o4-mini

	Complexity n				
Method	13	14	15	16	17
AD	3.18 $\pm$ 0.04	3.64 $\pm$ 0.03	4.40 $\pm$ 0.04	5.16 $\pm$ 0.04	5.91 $\pm$ 0.07
AD + voting	10.11 $\pm$ 0.20	11.07 $\pm$ 0.07	13.95 $\pm$ 0.29	15.86 $\pm$ 0.18	20.73 $\pm$ 0.51
LEAD	51.99 $\pm$ 1.55	53.63 $\pm$ 0.48	78.42 $\pm$ 2.80	88.83 $\pm$ 3.10	435.04 $\pm$ 1.23

Table 5: Estimated tokens per solution ($\times 10^5$) for GPT-5.2 model

G SOLUTION FORMAT

Listing 3: Expected output for Towers of Hanoi ($N = 3$)

```
solution = [
  {'move': [1, 0, 2], 'state': [[3, 2], [], [1]]},
  {'move': [2, 0, 1], 'state': [[3], [2], [1]]},
  {'move': [1, 2, 1], 'state': [[3], [2, 1], []]},
  {'move': [3, 0, 2], 'state': [[], [2, 1], [3]]},
  {'move': [1, 1, 0], 'state': [[1], [2], [3]]},
  {'move': [2, 1, 2], 'state': [[1], [], [3, 2]]},
  {'move': [1, 0, 2], 'state': [[], [], [3, 2, 1]]},
]
```

Listing 4: Expected output for Checkers Jumping ($N = 2$)

```
solution = [
  {'move': ['R', 0, 1], 'state': ['_', 'R', 'B']},
  {'move': ['B', 2, 0], 'state': ['B', 'R', '_']},
  {'move': ['R', 1, 2], 'state': ['B', '_', 'R']},
]
```

H PROMPTS

Listing 5: Atomic Decomposition Prompt used for the Checkers Jumping task

```
Here is the puzzle.

I have a one-dimensional board with 2*N+1 cells, where N red checkers ('R') on the le
The goal is to swap the positions of red and blue checkers, effectively mirroring the

Rules:
- Checkers can only move forward (towards the opposite side). More explicitly, red ch
- A checker can slide forward into an adjacent cell, provided that this cell is empty
- A checker can jump forward over one adjacent checker of opposite color, landing in

The positions of the board are indexed from 0 (the leftmost cell) to 2*N (the rightm
We represent each move as a list of the form [<checker_color>, <position_from>, <pos

Now, let's design a solution to this puzzle.

When we have a continuous block of the blue checkers adjacent to the left corner, the
Consequently, a blue checker is considered to be a part of a complete block, if all t
Similarly, a red checker is considered to be a part of a complete block, if all the

Now, let's find out the losing position (i.e. the one for which it's impossible to re
 - (P1): if we have ['_', 'R', 'R'] segment on the board, and these two red checkers
 - (P2): similarly, if we have ['B', 'B', '_'] segment on the board, and these two b
 - (P3): besides, if we have ['B', '_', 'R'] segment on the board, where neither the
Thus, when solving the puzzle, we have to avoid any move that leads to the positions

Finally, it turns out that any valid move that does not lead to any of the losing po

### Solution strategy
The following simple strategy will lead to the optimal solution of the puzzle: at eac
If there multiple moves satisfying this condition, select any of them.
One may show that as long as we follow this strategy, we will always have at least on

You DO NOT need to solve the puzzle.
Instead, you will be given an intermediate position of the game, and you have to find

### Efficient Algorithm for Finding an Optimal Move and Executing It

Here's how to efficiently compute optimal moves for any board configuration:

**Step 1: Find the empty position**
- Scan the board once to find the position of '_'
- `empty_pos`: The index (0-based) of the empty cell on the board

**Step 2: Identify movement patterns around the empty cell**
Check the following patterns in O(1) time by examining cells adjacent to the empty p

- `['R', '_']_pattern`: Boolean indicating if there's a red checker immediately to th
  - Check: `board[empty_pos - 1] == 'R'` (if `empty_pos > 0`)
- `['_', 'B']_pattern`: Boolean indicating if there's a blue checker immediately to t
  - Check: `board[empty_pos + 1] == 'B'` (if `empty_pos < len(board) - 1`)
- `['R', 'B', '_']_pattern`: Boolean indicating if there's a red checker that can jum
  - Check: `board[empty_pos - 2] == 'R'` and `board[empty_pos - 1] == 'B'` (if `empt
- `['_', 'R', 'B']_pattern`: Boolean indicating if there's a blue checker that can ju
```

```
  - Check: `board[empty_pos + 1] == 'R'` and `board[empty_pos + 2] == 'B'` (if `empt

**Step 3: Generate valid moves**
Based on the patterns above, construct the list of all valid moves:

- If `['R', '_']_pattern` is True -> add `['R', empty_pos - 1, empty_pos]` (red checi
- If `['_', 'B']_pattern` is True -> add `['B', empty_pos + 1, empty_pos]` (blue che
- If `['R', 'B', '_']_pattern` is True -> add `['R', empty_pos - 2, empty_pos]` (red
- If `['_', 'R', 'B']_pattern` is True -> add `['B', empty_pos + 2, empty_pos]` (blue

**Step 4: Filter optimal moves**
For each valid move, simulate the resulting board state and check if it creates any

For move `[color, from_pos, to_pos]`:

1. **Simulate the move:**
   - `next_state`: Copy of the current board with the move applied
     - Set `next_state[from_pos] = '_'` (source position becomes empty)
     - Set `next_state[to_pos] = color` (destination gets the checker)
   - `next_empty_pos`: Position of the empty cell after the move (equals `from_pos`)

2. **Find complete blocks in the resulting state:**
   - `next_left_block_end`: Rightmost position of the complete blue block from the le
     - Scan from left until first non-'B' cell, return position-1 (or -1 if no compl
   - `next_right_block_start`: Leftmost position of the complete red block from the
     - Scan from right until first non-'R' cell, return position+1 (or `len(board)`

3. **Check for losing patterns:**

   **Pattern P1 `['_', 'R', 'R']`:**
   - `p1_found`: Boolean indicating if the pattern `['_', 'R', 'R']` exists starting
     - Check: `next_empty_pos <= len(board)-3` and `next_state[next_empty_pos+1] ==
   - `is_red_complete`: Boolean indicating if the two red checkers in the pattern are
     - Check: `next_empty_pos+2 >= next_right_block_start` (if `p1_found` is True)
   - `p1_losing`: Boolean indicating if this pattern makes the position losing
     - Value: `p1_found and not is_red_complete`

   **Pattern P2 `['B', 'B', '_']`:**
   - `p2_found`: Boolean indicating if the pattern `['B', 'B', '_']` exists ending a
     - Check: `next_empty_pos >= 2` and `next_state[next_empty_pos-2] == 'B'` and `n
   - `is_blue_complete`: Boolean indicating if the two blue checkers in the pattern
     - Check: `next_empty_pos-2 <= next_left_block_end` (if `p2_found` is True)
   - `p2_losing`: Boolean indicating if this pattern makes the position losing
     - Value: `p2_found and not is_blue_complete`

   **Pattern P3 `['B', '_', 'R']`:**
   - `p3_found`: Boolean indicating if the pattern `['B', '_', 'R']` exists centered
     - Check: `next_empty_pos >= 1` and `next_empty_pos <= len(board)-2` and `next_s
   - `is_blue_complete`: Boolean indicating if the blue checker in the pattern is pa
     - Check: `next_empty_pos-1 <= next_left_block_end` (if `p3_found` is True)
   - `is_red_complete`: Boolean indicating if the red checker in the pattern is part
     - Check: `next_empty_pos+1 >= next_right_block_start` (if `p3_found` is True)
   - `p3_losing`: Boolean indicating if this pattern makes the position losing
     - Value: `p3_found and not is_blue_complete and not is_red_complete`

4. **Determine if move is optimal:**
   - The move is optimal if `not (p1_losing or p2_losing or p3_losing)`

**Step 5: Execute the first optimal move**
```

```
Once all valid moves have been filtered:

1. **Select the move:**
   - `selected_move`: The first move from the `optimal_moves` list
     - Format: `[color, from_pos, to_pos]`

2. **Execute the move:**
   - `new_state`: Copy of the current board
   - Set `new_state[from_pos] = '_'` (source position becomes empty)
   - Set `new_state[to_pos] = color` (destination gets the checker)

3. **Return the result:**
   - Return a dictionary with the selected move and the resulting board state
   - Format: `{'move': selected_move, 'new_state': new_state}`

You will be given a position of the board, and your task is to find an optimal move

For instance, assume N = 2 and the following position is given:

position = ['R', '_', 'R', 'B', 'B']

The solution for this query is the following:

solution = {'move': ['B', 3, 1], 'new_state': ['R', 'B', 'R', '_', 'B']}

### Task
Consider this puzzle for N = 3 and the following position of the board:

position = ['B', 'B', 'B', 'R', '_', 'R', 'R']

Find an optimal move for this position and the new state after executing this move.

Requirements
- Your final answer must be in the following format:

solution = {'move': <move>, 'new_state': <new state>}
```

Listing 6: Iterative Restart prompt ending used for the Checkers Jumping task

```
...

### Task
Consider this puzzle for N = 3 and the following position of the board:

position = ['R', 'R', 'R', '_', 'B', 'B', 'B']

Continue the solution from the given position.
If you can complete the solution, output the entire solution completion. Otherwise,

Requirements
- Your final answer must be in the following format:

moves = [
  {'move': [<checker_color>, <position_from>, <position_to>], 'state': <subsequent st
  ...
]

- Do not include any explanatory comments in the final output.
```

Listing 7: Atomic Decomposition prompt used for the Tower of Hanoi task

```
Here is the puzzle.

There are three pegs and N disks of different sizes stacked on the first peg from the
The disks are numbered from 1 (smallest) to N (largest). We denote the pegs as peg_0

Rules:
1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one stack and placing it on top
3. A larger disk may not be placed on top of a smaller disk.

The goal is to move the entire stack to the third peg.

We represent each step as a dict with the following items:
- "step_id": int, index of the current move.
- "move": a list of the form [disk_number, peg_from, peg_to]
  - disk_number -> the number of the disk being moved (1 = smallest)
  - peg_from -> the index (0-2) of the peg the disk is moved from
  - peg_to -> the index (0-2) of the peg the disk is moved to
- "state": a list of three lists representing the configuration of the pegs after the
  - Each inner list represents one peg (peg 0, peg 1, peg 2)
  - Each peg list contains the disks currently on that peg, ordered from bottom (on

You DO NOT need to solve the whole puzzle.
Instead, you will be given an intermediate position in the game (together witg the la

For instance, assume N = 3 and the following configuration is given:

configuration = {'step_id': 3, 'state': [[3], [2, 1], []]}

The solution for this query is the following:

solution = {'step_id': 4, 'move': [3, 0, 2], 'old_state': [[3], [2, 1], []], 'state'

### Iterative Solution for Tower of Hanoi

Here is the algorithm how to perform the next step in the solution, given the curren

configuration = {step_id: <step_id>, state: <state>}

Algorithm:

1. Calculate number of completed moves:
    num_moves = step_id

2. Find a correct move:

    If num_moves is even:
        - Find the peg which contains disk 1 on its top. Denote this peg as peg_i
        - Define the movement cycle for the smallest disk based on the parity of N (
            - If N is odd: cycle = [peg_0, peg_2, peg_1]
            - If N is even: cycle = [peg_0, peg_1, peg_2]
        - Let peg_j be the next peg after peg_i in cycle (in cyclic order, meaning th
        - Move disk 1 from peg_i to peg_j:
            move = [1, peg_i, peg_j]

    If num_moves is odd:
        - Sort the three pegs by the size of their top disk in descending order (trea
```

```
            Denote sorted pegs as peg_i, peg_j peg_k (meaning that peg_i has the larg
            Let d be the number of the top disk on peg_j.
        - Move disk d from peg_j to peg_i:
            move = [d, peg_j, peg_i]

3. Execute the move:
    - Let the found move be
        move = [<disk_number>, <peg_from>, <peg_to>]
    - Update state:
        - create new_state as a copy of state
        - remove the last element from new_state[peg_from] list
        - append <disk_number> to new_state[peg_to] list

4. Update the step_id:
    - new_step_id = step_id + 1

5. Output the new step:
    solution = {'new_step_id': new_step_id, 'move': move, 'new_state': new_state}


# Task:
Consider this puzzle for N = 4 and the following configuration of the game:

configuration = {step_id: 0, state: [[4, 3, 2, 1], [], []]}

Requirements:
- Your final answer must be in the following format:

solution = {'new_step_id': <new_step_id>, 'move': <move>, 'new_state': <new_state>}
```