DISC: Dynamic Decomposition Improves LLM Inference Scaling

Jonathan Light[1,2,4*], **Wei Cheng**[2✉], **Benjamin Riviere**[4], **Yue Wu**[3], **Masafumi Oyamada**[5], **Mengdi Wang**[3], **Yisong Yue**[4], **Santiago Paternain**[1], **Haifeng Chen**[2]
[1]Rensselaer Polytechnic Institute, [2]NEC Laboratories America, [3]Princeton University,
[4]California Institute of Technology, [5]NEC Corporation

Abstract

Inference scaling methods for LLMs often rely on decomposing problems into steps (or groups of tokens), followed by sampling and selecting the best next steps. However, these steps and their sizes are often predetermined or manually designed based on domain knowledge. We propose dynamic decomposition, a method that adaptively and automatically partitions solution and reasoning traces into manageable steps during inference. By more effectively allocating compute – particularly through subdividing challenging steps and prioritizing their sampling – dynamic decomposition significantly improves inference efficiency. Experiments on benchmarks such as APPS, MATH, and LiveCodeBench demonstrate that dynamic decomposition outperforms static approaches, including token-level, sentence-level, and single-step decompositions, reducing the pass@10 error rate by 5.0%, 6.7%, and 10.5% respectively. These findings highlight the potential of dynamic decomposition to improve a wide range of inference scaling techniques.

1 Introduction

Scaling inference efficiency remains a fundamental challenge for large language models (LLMs). Many existing approaches improve inference by decomposing problems into smaller steps and systematically exploring different solutions [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. Some decomposition methods often rely on domain-specific heuristics and hand-crafted rules [11, 12, 13]. However, manually partitioning problems or designing task-specific heuristics is costly and lacks generalization. Moreover, identifying critical steps for an LLM can be non-trivial for humans. LLMs may assign importance to seemingly trivial words (e.g., therefore or which), which, while counterintuitive to humans, play a crucial role in autoregressive generation [14]. Other approaches employ fixed, uniform step sizes, such as token- or sentence-level decomposition [1, 15]. All these methods rely on **static decomposition strategies**, where step sizes are predefined or determined via heuristics. Such rigidity risks overusing compute on steps that are easy for the LLM (but potentially difficult for humans) while undersampling more challenging steps.

To overcome these limitations, we propose DISC (Dynamic decomposition Improves Scaling Compute), a recursive inference algorithm that dynamically partitions solution steps based on difficulty. Unlike prior methods, DISC **adapts decomposition granularity** during inference based on both the available budget and problem complexity, ensuring finer granularity for more difficult steps. By leveraging the autoregressive nature of LLMs, DISC efficiently **locates difficult steps** through dynamically proposing step sizes, focusing compute on challenging regions rather than wasting resources on trivial steps. DISC is generalizable, requires *no human supervision, domain-specific heuristics, prompt engineering, or process annotations*, and is easily *parallelizable*, making it widely

[*]Work done during the part-time internship at NEC Laboratories America. [✉]Corresponding author.

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

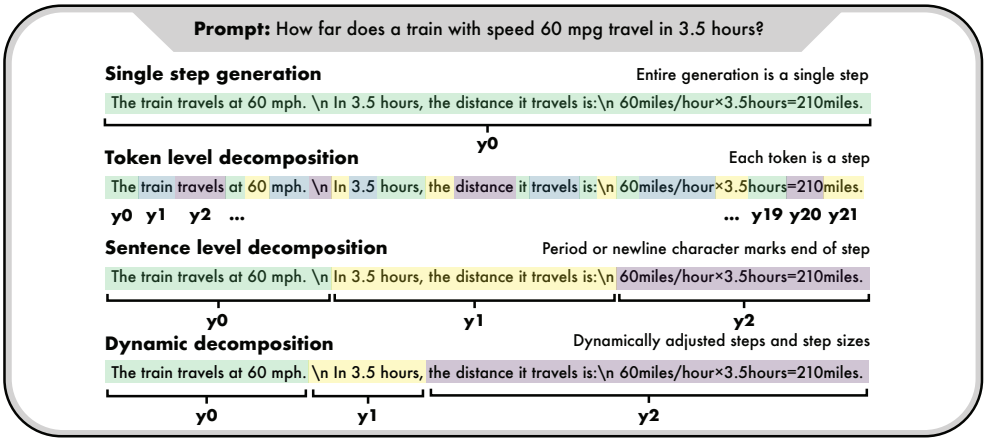

Figure 1: **Comparison of automatic decomposition strategies based on step size.** Coarser steps accelerate the search process but risk skipping over optimal solutions and committing to suboptimal prefixes. In contrast, finer steps ensure more precise decisions but lead to slower search. A dynamic strategy that adapts step size based on LLM feedback offers a balanced approach, combining the efficiency of coarse steps with the precision of fine-grained decomposition.

applicable across tasks. Furthermore, DISC is plug-and-play with off-the-shelf search algorithms and can be naturally integrated with greedy, beam, or Monte Carlo Tree Search.

Our main contributions are:

- We introduce **DISC**, a method for dynamically adjusting step sizes and decomposing solutions during inference without human supervision, domain-specific heuristics, or process reward models.
- We demonstrate how DISC integrates decomposition directly into inference-time search, **allocating compute more effectively toward high-potential solution prefixes**.
- We show that DISC improves inference scaling in terms of both **sample efficiency**, **token efficiency**, and **runtime**, achieving up to **10% reduction** in error relative to the baselines and up to **4x increase** in accuracy over the base model, with just 10 samples, including with reasoning models.
- We provide both empirical and theoretical insights into LLM reasoning, including identifying **critical intermediate steps** and analysis of how DISC helps discover optimal solutions.

2 Preliminaries

2.1 Problem Setting

We consider a reasoning and code generation setting where we are given: a dataset $\mathcal{X} = \{\boldsymbol{x}^{(i)}\}_{i=1}^{N}$, a pretrained, autoregressive LLM π that generates solutions $\boldsymbol{y} \in \mathcal{Y}$, and a reward model $R : \mathcal{X} \cdot \mathcal{Y} \to [0, 1]$ that evaluates the generated solutions. The goal is to find inputs to the LLM that produce solutions with high reward. This setting includes program synthesis, where correctness is verified using ground-truth tests [16, 17], and mathematical reasoning, where solutions are validated numerically [18, 19]. The reward model can be a ground-truth verifier, a trained heuristic [20], self-consistency [21], or an LLM-as-a-judge [22] and because our focus is on decomposition rather than verification, we use the ground-truth reward model where available.

We use the following hierarchical token notation: y is a token, $\boldsymbol{p}$ is a prefix that starts at the prompt and concatenates multiple steps, $\boldsymbol{s}$ is a suffix that concatenates multiple steps and ends with the EOS token, and $\boldsymbol{y}$ is a complete solution that starts at the prompt, concatenates multiple steps, and ends at the end of sequence token EOS. We denote a concatenation of two tokens or token sequences with $\cdot$, e.g. $\boldsymbol{p} \cdot \boldsymbol{s}$ is the concatenation of prefix $\boldsymbol{p}$ and suffix $\boldsymbol{s}$ to form a complete solution $\boldsymbol{y}$. We denote the sampled suffix from prefix $\boldsymbol{p}$ using the LLM policy as: $\boldsymbol{s} \sim \pi(\cdot|\boldsymbol{p})$. We denote an optimal solution with $\boldsymbol{y}^*$ and an optimal suffix as $\boldsymbol{s}^*$, where there exist multiple optimal solutions. For our analysis, we use the convention that $\pi(\boldsymbol{s}^*|\boldsymbol{p}) = 0$ if there does not exist a completion $\boldsymbol{s}^*$ such that $\boldsymbol{p} \cdot \boldsymbol{s}^*$ is optimal. The **size** of a string $|\boldsymbol{y}|$, refers to its length in tokens or characters.

2.2 Existing Decomposition Methods

Single-step generation. In a single-step generation, the entire solution is generated in one pass from the prompt to the EOS token, treating it as a single action. This approach underlies the widely used inference scaling method **best of n** (BoN) [19, 23, 5, 24], where n complete solutions are sampled,

and the highest-scoring one is selected. Single-step generation also plays a role in alignment and fine-tuning methods such as **DPO** [25] and **RLOO** [26].

Token-level decomposition. At the opposite end of the spectrum, token-level decomposition treats each atomic token as an individual step. While this approach dramatically increases search complexity, it enables fine-grained search that can yield higher performance gains given sufficient compute [1].

Newline and sentence-level decomposition. A commonly used decomposition method segments LLM generations into sentences or lines based on delimiters such as periods or newlines [27, 1, 11]. Typically, each newline corresponds to a new paragraph, equation, or line of code, which often encapsulates a distinct reasoning step.

Challenge: Automatic and scalable decomposition

Existing decomposition methods are static and manually designed, resulting in either slow convergence to good performance or fast convergence to poor performance. We propose adaptive decomposition for fast convergence to a good performance.

2.3 Search for Inference Scaling

We differentiate between decomposition and search, where decomposition controls the number of steps between new branches and search is the process of selecting which branch to explore. In other words, decomposition is the construction of nodes and edges, and search is a process that occurs on that structure. In this work, we propose a decomposition method that is plug-and-play with different search methods. In our experimental results in Sec. 4.1, we compare decomposition methods: token-level decomposition, sentence-level decomposition, and DISC with the search methods: greedy, beam [28], and Monte Carlo Tree Search [1, 29]. Implementation details are provided in App. E.

3 Methodology

The DISC algorithm uses LLM rollout data to dynamically decompose reasoning steps, adjust step sizes, and allocate sampling compute. In Sec. 3.2, we present DISC paired with a greedy search strategy, shown in Alg. 1 and Fig. 2. In Sec. 3.4, we present the general case of DISC as a nodal expansion operator which is used for pairing with beam search and MCTS.

3.1 High-level Overview

At a high level, DISC with Greedy Search advances a base prefix $\boldsymbol{p}_b$ forward by iteratively concatenating promising steps to it. The core intuition is to dynamically allocate compute by adjusting the step size: if a prefix shows promising reward improvement, we take a large step forward; if not, we contract the step and concentrate sampling around that prefix. This adaptive behavior focuses the LLM's effort on regions of the search space that are more likely to yield high-reward completions.

Fig. 2 illustrates a single iteration of this decision process, where a candidate prefix is either accepted and extended or rejected and contracted. Over multiple iterations, this process yields a full solution composed of several such accepted steps. Fig. 3 shows how the prefix is incrementally constructed: it displays the number of steps DISC has committed to the prefix, and how the prefix used for sampling dynamically changes over time. Together, these figures highlight the local step-wise decision-making and the global trajectory of prefix refinement throughout the search.

Algorithm 1 DISC with Greedy Search

Require: LLM π, Reward model R, prompt x, initial partition fraction α_0, negative binomial threshold σ, sample budget N

// initialize base prefix and current partition fraction

$\boldsymbol{p}_b = \boldsymbol{x}$, $\alpha = \alpha_0$, $n = 0$

// compute base prefix statistics (_b)

$Y_b = \{(\boldsymbol{p}_b \cdot \boldsymbol{s}^i, R(\boldsymbol{s}^i)) \mid \boldsymbol{s}^i \sim \pi(\cdot|\boldsymbol{p}_b)\}_{i=1}^{M}$

$z_b = (\max(Y_b) - \text{mean}(Y_b))/\text{std}(Y_b)$

$(\boldsymbol{p}_b \cdot \boldsymbol{s}_b^*) = \text{argmax}(Y_b)$

while $n < N$ **do**

  // get candidate prefix (_c)

  $\boldsymbol{p}_c = \boldsymbol{p}_b \cdot \text{split}(\boldsymbol{s}_b^*, \alpha)$

  // sample and compute candidate prefix statistics (_c)

  $Y_c = \{(\boldsymbol{p}_c \cdot \boldsymbol{s}^i, R(\boldsymbol{p}_c \cdot \boldsymbol{s}^i)) \mid \boldsymbol{s}^i \sim \pi(\cdot|\boldsymbol{p}_c)\}_{i=1}^{M}$

  $z_c = (\max(Y_c) - \text{mean}(Y_c))/\text{std}(Y_c)$

  $(\boldsymbol{p}_c \cdot \boldsymbol{s}_c^*) = \text{argmax}(Y_c)$

  $n \leftarrow n + M$

  // accept or reject the candidate prefix

  if $z_c < z_b$ or $|\boldsymbol{s}_b^*| \leq 1$ **then**

    $\boldsymbol{p}_b \leftarrow \boldsymbol{p}_c$, $\boldsymbol{s}_b^* \leftarrow \boldsymbol{s}_c^*$,

    $\alpha \leftarrow \alpha_0$, $z_b \leftarrow z_c$

  else

    $\alpha \leftarrow \alpha_0 \alpha$

  end if

end while

yield $\boldsymbol{p}_b \cdot \boldsymbol{s}_b^*$

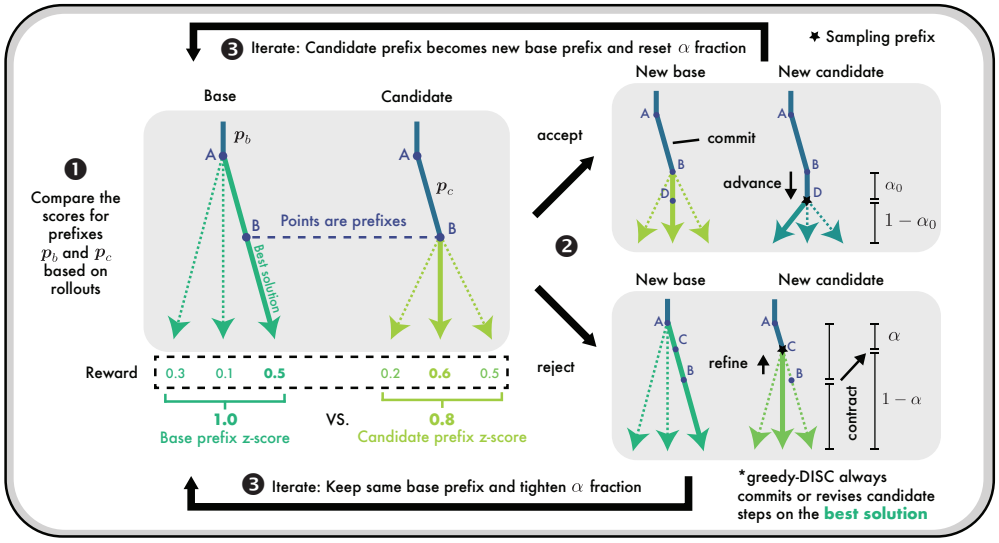

Figure 2: **DISC with Greedy Search.** One iteration of Alg. 1. We start with a base prefix A and a candidate prefix B. We compare the sample statistics of each by evaluating a scoring function (e.g., z-score). If the candidate prefix B demonstrates a higher likelihood of reward improvement compared to continuing from base prefix A, we accept B, commit it as the new base, and extend the candidate to a further step (e.g., BD) on the best sampled solution. If B is not better, we reject it and propose a shorter candidate (e.g., AC), contracting the step size. This process repeats until a new candidate is accepted or all options are exhausted. The algorithm thus adaptively advances or contracts the step size and search horizon based on the relative quality of completions from each prefix.

Assumptions. DISC makes minimal, broadly applicable assumptions, enabling generality and ease of deployment. It avoids handcrafted prompts, process-level reward models, and domain-specific heuristics. Its only requirement is access to a scalar outcome reward model (ORM) to guide search. In the absence of a ground-truth ORM, self-supervised signals—like LLM critiques or unit tests—serve as effective substitutes [30, 31, 32]. It also assumes the underlying policy π can generate continuations from any prefix $\boldsymbol{p}$, a standard feature of decoder-only language models.

3.2 DISC with Greedy Search Algorithm

We now describe Alg. 1. The algorithm takes as input an LLM π, a reward model R, prompt $\boldsymbol{x}$, initial partition fraction α_0, threshold σ, and sample budget N. It initializes the base prefix as $\boldsymbol{p}_b = \boldsymbol{x}$ and sets $\alpha = \alpha_0$.

At each iteration, the algorithm samples π from $\boldsymbol{p}_b$ to generate completions $\boldsymbol{y}^i = \boldsymbol{p}_b \cdot \boldsymbol{s}^i$, computing rewards $R(\boldsymbol{y}^i)$. Sampling stops when the cumulative reward exceeds σ: $M = \min\{m \in \mathbb{Z}_{>0} \mid \sum_{i=1}^{m} R(\boldsymbol{y}^i) \geq \sigma\}$. This balances sample quantity and quality. The solutions and their rewards are stored in Y, and the best suffix $\boldsymbol{s}^*$ and z-score are computed.

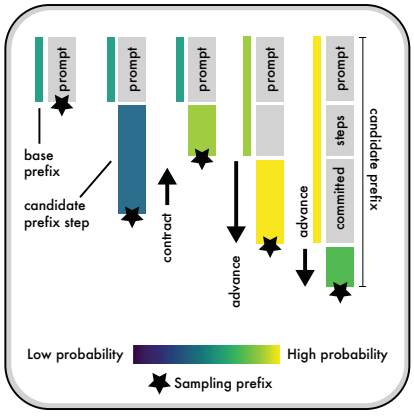

Figure 3: **Multiple iterations of Alg. 1.** DISC dynamically refines its step sizes across iterations, advancing and contracting the prefix at which it samples from.

A candidate prefix $\boldsymbol{p}_c$ is then formed by appending the first α fraction (token-wise) of $\boldsymbol{s}^*$ to $\boldsymbol{p}_b$. It is accepted if its reward z-score z_c is lower than the current z_b. Assuming rewards follow a location-scale family distribution (e.g., Gaussian), a lower z-score implies a higher probability of improvement, since $\Pr(R > \max(Y_{b,c})) = 1 - \mathrm{CDF}(z_{b,c})$ where CDF is the cumulative distribution function. Fig. 4 displays how we can estimate the probability of improvement from a reward distribution. A location-scale distribution of rewards is supported empirically (see App. C.5).

If the candidate is accepted, the algorithm updates the base prefix $\boldsymbol{p}_b \leftarrow \boldsymbol{p}_c$, resets $\alpha = \alpha_0$, and updates the base z-score and best suffix. If rejected, the partition fraction contracts: $\alpha \leftarrow \alpha \cdot \alpha_0$.

This contraction implements DISC's recursive refinement mechanism, focusing the search on more promising regions. The process repeats until the sample budget is exhausted or a correct solution is found.

3.3 Analysis of DISC with Greedy Search

DISC exhibits two important properties: (i) the z-score decreases monotonically over the course of algorithm iterations, and (ii) the best candidate solution always has a higher reward than the best base solution. We leverage these properties, together with assumptions about the quality of π, to establish the following result. We also develop a motivating theoretical example in App. F.2.

Theorem 1 (Optimality of DISC) *Consider Alg. 1. Suppose that for some problem $\boldsymbol{x}$, the optimal solution is in the support of $\pi(\cdot \mid \boldsymbol{x})$. Then at some $n > 0$, the base prefix contains* EOS *token, the algorithm terminates, and this solution is an optimal solution. See App. F for proof.*

Remark 1 *Our analysis is dependent on a strong policy assumption, which is "reverse engineered" to be the weakest assumption on the policy such that our algorithm terminates at optimality. In other words, this assumption depends on an instance-dependent property that must be checked empirically. The purpose of our analysis is not to provide a universal guarantee, but rather to understand how adaptive decomposition method can control real-time inference compute without sacrificing optimality.*

3.4 DISC for Plug-and-Play with Search Algorithms

In the previous section, we presented DISC with greedy search as a complete algorithm. However, the core of DISC is the decomposition that controls from which token prefixes to query the model and dynamically adjusts step sizes, which can be plug-and-played with other search algorithms like Beam search and MCTS. The DISC EXPANDNODE and MAKECHILDREN are presented in Alg. 3. These methods are conceptually similar to the DISC with greedy search, except that in the general search context, each node stores its own base prefix, and so when a node is expanded it computes its candidate prefix from the suffix to that node instead of from the best existing suffix. During the expansion operator, multiple sets of children are generated, but only the children from the final prefix are kept for search.

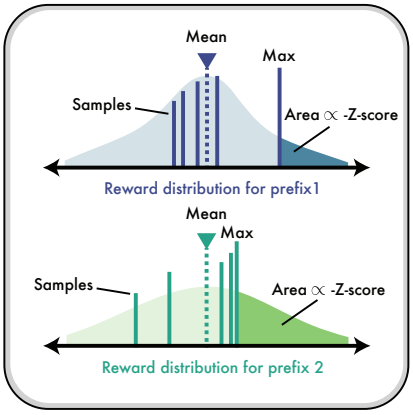

Figure 4: Reward distribution of rollouts sampled from two different prefixes. The probability of sampling a higher rollout from prefix 2 is higher than that of prefix 1.

3.5 Example Decomposition

Using a sampling budget of $N = 100$ LLM calls, the decomposition of a representative MATH problem is shown below. Each step is enclosed in brackets and color-coded based on the z-score of sampling a better suffix conditioned on the current prefix. Brighter colors indicate lower z-scores, signaling higher importance.

DISC example decomposition

[Let the length of the rectangle be l] [and the width of the rectangle be w. Since the perimeter of the rectangle is 24 inches, we have that $2l + 2w = 24$, so $l + w = 12$. We wish to maximize the area of the rectangle,] [which] [is $A = lw$. Let $l = 12 - w$ and plug into the area:

$$A = (12 - w)w \Rightarrow \qquad A = 12w - w^2$$

Now, we differentiate A with respect to w:

$$A'(w) = 12 - 2w$$

We wish to maximize A, so we set $A'(w) = 0$, and solve for w:

$$12 - 2w = 0 \Rightarrow \qquad w = 6$$

Since $l = 12 - w$, we have that $l = 12 - 6 = 6$. Therefore, the area of the rectangle is $A = lw = 6 \cdot 6 = \boxed{36}$.]

DISC identifies the **first step** as highly important, which aligns with intuition—early reasoning forms the foundation for all subsequent steps. In contrast, the **final step**, although large, is marked as low-importance, indicating that DISC allocated minimal compute toward refining it. This suggests that once earlier reasoning is fixed, there is limited opportunity for improvement in the final conclusion using additional sampling. Interestingly, the **third step**, beginning with *"which"*, is assigned high importance and receives substantial attention from DISC. This step appears to act as a pivotal decision point that shapes the direction of the remaining solution. This observation supports the idea that certain tokens or sub-sequences function as critical reasoning forks or pivots—consistent with findings from prior work [33, 34].

Autoregressive models require autoregressive decomposition

While transition words such as 'which', 'therefore', 'wait', etc. may not appear significant to human readers, our decomposition frequently identifies them as critical decision points for autoregressive LLMs trained on next-token prediction, where selecting a different token at these junctures can substantially alter the downstream reasoning and final outcome. Therefore, it is essential for inference algorithms to allocate more compute toward sampling at these steps, and to identify these steps automatically through LLM data rather than through human design.

4 Experimental Results

4.1 Main Results

Benchmarks. We evaluate DISC on three benchmarks: **APPS**, **MATH**, and **LiveCodeBench**, to assess its impact on inference scaling for both coding and reasoning. **APPS** [18] consists of 5000 competitive programming problems across three difficulty levels, with the competition-level subset being the hardest. We evaluate on a 200-problem subset due to computational constraints. **MATH** [35] comprises 12,500 math problems. Since the ground-truth verifier provides only binary rewards, we use a pretrained ORM [36], trained via the method in [37], with Llama-3.1-8B-Instruct as the base model. We test on a 500-problem subset (**MATH500**), identical to prior work [37, 23]. **LiveCodeBench** [38] is a continuously updated dataset from Leetcode, AtCoder, and CodeForces, ensuring LLMs have not been exposed to test problems. We evaluate on the 108 problems uploaded between 10/01/2024 and 12/01/2024 to prevent contamination.

Baselines. We compare DISC against three prior decomposition methods: **TokenSplit** (token-level decomposition), **LineSplit** (newline-based decomposition), and **BoN** (treating the entire solution as a single step). These are all standard and the most commonly used methods (see Sec. 2.2).

Metrics. We evaluate two key metrics: **Pass@k**, the proportion of problems solved within a sample budget k, and **Pass@token**, the proportion solved within a given token budget. Note that k refers to the sample budget, *not thousands of samples*, and error proportion refers to proportion not solved. We use $\alpha_0 = 0.15, \sigma = 1.0$, and temperature $\tau = 0.2$ by default for DISC unless otherwise specified.

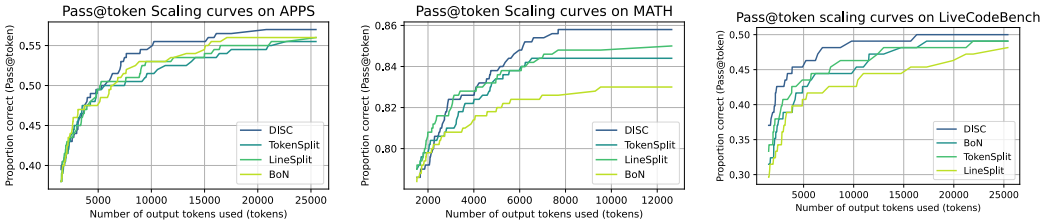

Figure 5: **Token-level comparisons across benchmarks** using gpt-4o-mini. (Left) APPS competition level (Middle) MATH500 (Right) LiveCodeBench. DISC achieves superior inference scaling over baselines on all three benchmarks.

Performance. Across all benchmarks, DISC consistently delivers stronger performance and better scaling under both fixed token budgets (Fig. 5) and fixed sample budgets (Fig. 7). For example, on APPS, the pass@10 error proportion decreases from 0.50 to 0.475; on MATH500, from 0.15 to 0.14; and on LiveCodeBench, from 0.57 to 0.51. These correspond to a **5.0%, 6.7%, and 10.5% reduction in error** relative to the best baseline, respectively. These improvements are particularly meaningful on more challenging benchmarks, where performance gains are harder to achieve, demonstrating DISC 's effectiveness in guiding search toward high-reward regions. Extended results and analyses are provided in App. D.1, D.4, and D.5.

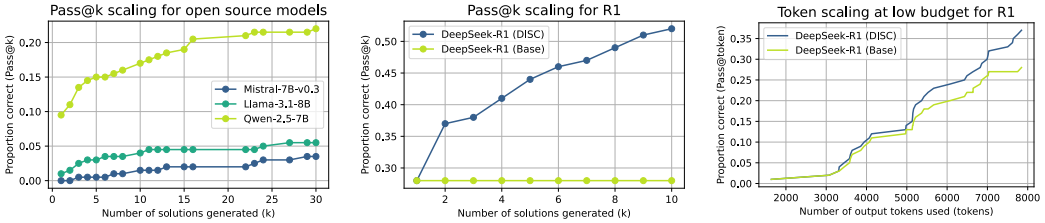

Figure 6: **Inference scaling for different models, including reasoning models, on APPS.** (Left) Pass@k for open-source models (Middle) Pass@k for R1 (Right) Pass@token for R1. DISC almost **doubles** base performance across all models.

Model Generality. DISC also provides substantial gains across a range of models, including open-source LLMs. For instance, it improves LLaMA's pass@10 rate from 0.01 to 0.04—a **300% relative increase**. Similarly, it boosts Mistral's performance from 0.0 to 0.02, and Qwen's from 0.095 to 0.17, representing a **79% relative increase**. As shown in Fig. 6 (left), these improvements hold consistently across sampling budgets, including in low-budget regimes. This demonstrates DISC 's general applicability and effectiveness even for weaker or resource-constrained models. Additional results and analyses are provided in App. C.3.

Reasoning Models. DISC also yields substantial gains when applied to strong long-form chain-of-thought (CoT) reasoning models such as R1 [15, 39]. As shown in Fig. 6 (middle), DISC improves accuracy by over **85%** relative to the base R1 model using just 10 samples. Notably, Fig. 6 (right) shows that even under a constrained token budget—matched to that of a single sample from the base model—DISC still achieves over a **33%** relative improvement. This demonstrates that DISC not only scales well with more samples but is also highly effective at identifying and prioritizing critical reasoning steps, leading to stronger performance even in low-resource settings.

Computational Overhead. DISC also produces negligible runtime overhead as shown in Fig. 43. Most of the overhead is still spent on token sampling, and as shown in Fig. 5 DISC is more token efficient than the baselines.

Self Generated Validation. We also evaluate DISC in a self-generated validation setting where a ground-truth reward model is unavailable [30, 40, 41]. Instead of relying on predefined test cases, we prompt the LLM to generate validation test cases based on the problem prompt. In real-world applications, manually curated ground-truth test cases are often costly to obtain, making self-generated validation a more scalable approach. The results, shown in Fig. 7, indicate that DISC

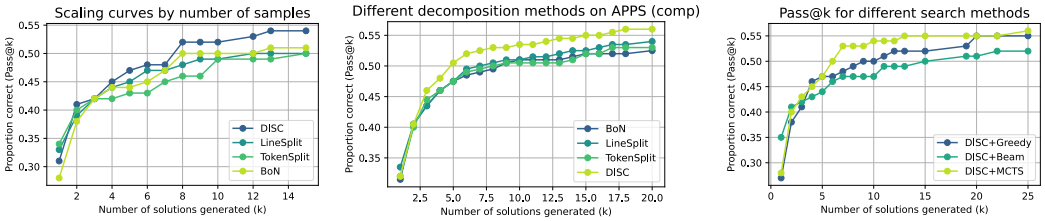

Figure 7: **Comparison of Pass@k performance on APPS using gpt-4o-mini.** (Left) self generated validation tests, (Middle) with ground truth tests, (Right) with different search methods.

continues to scale better than other methods in this setting, achieving a **54% relative increase in performance** over the base model. Additional results and details are provided in App. D.3.

Search. We demonstrate that search strategies such as **MCTS** and **beam search** can be naturally integrated with DISC using the approach described in Sec. 3.4. As shown in Fig. 46, greedy search tends to explore deeper partitions within the same search budget due to its myopic nature, while MCTS and beam search explore more diverse but shallower paths. Despite similar depth, **MCTS** outperforms beam search by allocating its search budget more strategically—focusing exploration on more promising candidates—resulting in superior overall performance, as seen in Fig. 7. Furthermore, unlike greedy search, which irrevocably commits to prefixes as they are accepted, MCTS maintains flexibility by exploring committing multiple candidate prefixes in parallel. Additional details and analysis are provided in App. E.

4.2 Ablation Studies

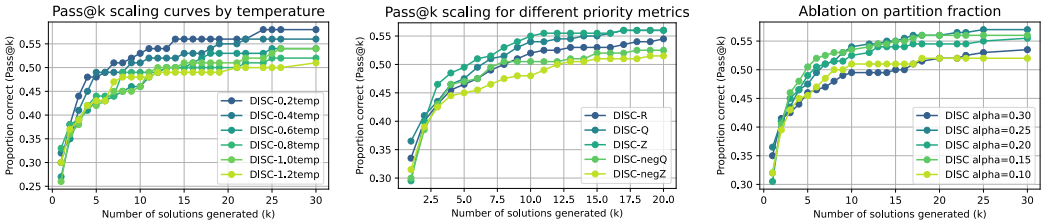

Figure 8: **Ablation study of DISC on APPS with gpt-4o-mini.** (Left) Effect of temperature: unlike BoN and other inference scaling methods, DISC achieves higher performance at lower temperatures. (Middle) effect of acceptance method (Right). Effect of partition fraction α_0: The range $0.15 \leq \alpha_0 \leq 0.25$ appears optimal.

Temperature. Typically, inference scaling methods achieve optimal performance at temperatures around 0.6–0.8, as increased temperature promotes sample diversity [42]. Surprisingly, however, DISC performs *better at lower temperatures*, as shown in Fig. 8. This trend is in stark contrast to BoN (Fig. 15), where higher temperatures are generally beneficial. We believe this phenomenon arises because DISC depends on estimating the z-score at each step using sample statistics. Lower temperatures reduce sample variance, leading to more reliable estimates, which in turn improves step selection. This is further supported by Fig. 13, which shows that lower temperatures yield lower standard deviations per step, indicating increased sampling consistency. Additional details and analyses can be found in App. C.1.

Acceptance Method. We perform an ablation study to evaluate whether using the z-score is an effective criterion for accepting candidate prefixes. Specifically, we compare our standard z-score-based acceptance method, DISC-Z, against four alternative baselines: DISC-R, which accepts candidates uniformly at random; DISC-Q, which accepts if the candidate prefix has a lower mean value; DISC-negQ, which accepts if the candidate has a higher mean; and DISC-negZ, which accepts if the candidate has a higher z-score (rather than a lower one). As shown in Fig. 8, the choice of acceptance criterion substantially influences performance. Among all methods, DISC-Z achieves the highest performance, while DISC-negZ performs worse than random selection, underscoring the importance of prioritizing candidates with a higher probability of improvement. Additional details and analysis are in App. C.2.

Partition Fraction α_0. As shown in Fig. 8 and Fig. 24, performance is highest when the partition fraction lies in the range $0.15 \leq \alpha_0 \leq 0.25$. Smaller values of α_0 generally yield better results because they lead to more conservative proposals—i.e., shorter candidate prefixes. This conservatism is beneficial due to the high cost of prematurely committing to a suboptimal prefix: once a candidate prefix is accepted (in greedy-DISC), it becomes fixed and cannot be revised. By keeping candidate prefixes short, the algorithm retains more flexibility to correct course in future steps. Additional analysis is provided in App. C.4.

4.3 Analysis and Interpretation

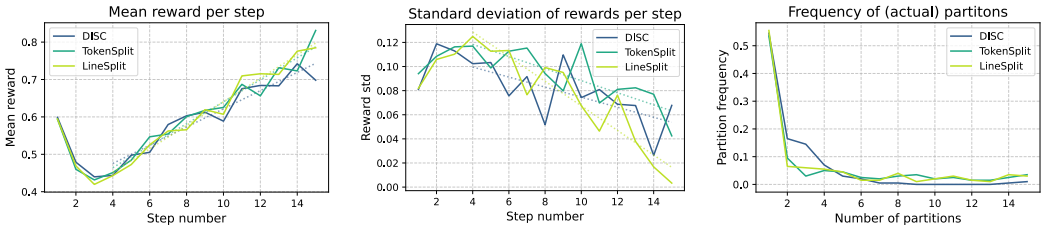

Figure 9: **Analysis of decomposition methods. (Left)** Average reward per step: From step 3 onward, higher step counts strongly correlate with increased average reward, demonstrating the effectiveness of decomposition. The dip between steps 1 and 3 likely occurs because simple problems are solved early, preventing further search. **(Middle)** Standard deviation of rewards per step: Decomposition reduces sampling variance, improving precision at deeper search depths. **(Right)** Frequency of the number of partition steps that the search algorithm committed to the prefix during the search process.

Our results strongly suggest that decomposition—whether line-based, token-based, or DISC —consistently improves sample quality. Fig. 9 (left and middle) shows how the mean and variance of sampled rewards evolve with the **step number**, i.e., the number of steps committed to the prefix during search. As the step number increases, the mean reward improves, indicating that longer committed prefixes lead to higher-quality solutions. At the same time, the variance of rewards decreases, suggesting that committing to a longer prefix also improves the precision of sampling. These trends highlight the benefits of finer-grained decomposition and incremental commitment in guiding the search process more effectively.

Furthermore, DISC achieves higher performance using fewer committed prefix steps—and thus fewer sampling stages—under a fixed sampling budget. Fig. 9 (right) shows the distribution of **actual partitions**, i.e., the number of steps effectively committed under a finite budget. As shown, DISC typically requires only 1–5 actual partition steps, whereas other methods commit to significantly more. This indicates that DISC is more efficient at identifying high-impact prefixes, enabling better performance with fewer sampling decisions.

Decomposition and Sample Quality

DISC enables more efficient exploration by identifying high-impact prefixes with fewer steps. Incremental prefix commitment not only improves sample quality—yielding higher average rewards—but also reduces reward variance, leading to more stable and reliable outputs under a fixed sampling budget.

5 Related Work

Inference scaling. Inference scaling has emerged as a dominant paradigm, driven by the introduction of o1- and r1-like chain-of-thought reasoning models [5, 6, 43, 44]. Several works examine the trade-off between inference compute and training compute [45, 46]. LLM inference often relies on decomposing complex problems into intermediate reasoning steps, as seen in chain-of-thought (CoT) prompting [47, 48, 49] and its variants [50, 51, 52, 53]. We extend inference scaling by introducing a new approach for adaptive compute allocation [43].

LLM reasoning and code generation. LLM reasoning and code generation are central tasks for inference scaling. Evolutionary inference scaling methods have been explored in program generation [54, 55, 56, 57, 58]. Domain-specific decomposition strategies have been applied in code generation, such as function-based decomposition [59, 60, 61]. More broadly, decomposition often

involves prompting LLMs to generate subtask completions [62, 63, 64], which differs from methods that refine a single LLM generation.

Reinforcement learning and Monte Carlo methods. Unlike standard RL, our setting resembles a search problem where the goal is to identify the single highest-reward path. Nested Monte Carlo search can accelerate optimal pathfinding [65]. Under the bandit setting, this can be formulated as identifying the arm with the highest *maximum* reward rather than the highest mean reward [66, 67].

6 Conclusion

We introduce DISC, a dynamic decomposition framework that adaptively partitions solution steps based on first order statistics that capture potential for improvement, improving inference scaling by directing compute toward critical steps while balancing exploration and resource allocation. DISC naturally integrates with search-based methods such as MCTS and beam search, further enhancing performance. It also identifies challenging steps for LLMs, aiding curriculum learning, fine-tuning, and dataset augmentation. By dynamically adjusting partitioning and step sizes based on available compute, DISC enables more adaptive and efficient reasoning in large language models, with broad implications for both training and inference optimization.