Test-Time Scaling for Scientific Equation Discovery

Haowei Lin* **Hubert Lim***
Xiangyu Wang **Letian Huang** **Di He**
Peking University
{linhaowei, di_he}@pku.edu.cn, HubertLinHong@stu.pku.edu.cn

Abstract

Test-time scaling (TTS) improves language model reasoning by allocating additional test-time compute, but prior work mainly studies closed-ended tasks such as math and coding. We study TTS for automated equation discovery, an open-ended setting where models search over candidate equations and rely on observed datapoints for feedback. We formulate LLM-driven equation discovery as an iterative search process that unifies Best-of-N, sequential refinement, tree search, and evolution-style methods under a common compute-allocation view. To isolate allocation effects from prompt engineering and other heuristics, we compare minimal parallel controllers under fixed budgets. On LLM-SRBench equation-discovery tasks, we find that search width is the dominant allocation parameter: the best width in our sweep generally increases with the compute budget, while the population–branching split and controller choice matter less. Appropriate width selection also improves wall-clock efficiency by increasing parallelism. These results suggest that, given an informative verifier, controlling exploration and exploitation is central to scaling LLM-based equation discovery.

1 Introduction

Test-time scaling (TTS) has become a standard way to improve the reasoning performance of language models by allocating additional compute at inference time rather than updating model parameters (Song et al., 2025; Snell et al., 2024). Most prior work studies closed-ended reasoning tasks such as mathematics and code generation, where the objective is to recover a correct answer to a well-specified problem (Guan et al., 2025; Li et al., 2025). In these settings, TTS is often framed as a question of *how much* extra inference-time compute to spend (Snell et al., 2024), or how more compute can help smaller models approach the performance of larger ones (Wu et al., 2024).

In this paper, we study TTS for automated equation discovery. Equation discovery is narrower than scientific discovery as a whole, but it captures an important open-ended modeling problem: a system must propose candidate equations or programs, receive feedback from an external evaluator, and iteratively search for better candidates. Unlike closed-ended reasoning benchmarks, there is typically no single target string to recover during search; progress is measured by improvement under a task-specific verifier. This makes equation discovery a useful testbed for studying how inference-time compute should be allocated across exploration and refinement.

This shift in perspective matters because recent LLM-based discovery and program-search systems already hint that the answer is not obvious. A growing line of work uses language models to generate candidate artifacts and closes the loop with an external verifier (Romera-Paredes et al., 2024; Novikov et al., 2025; Lange et al., 2025). These systems often work well, but they are usually introduced as complicated agent workflows with many coupled design choices, including specialized prompts, mutation rules, model ensembles, and hand-tuned pruning heuristics. As a result, it is difficult to tell which ingredients are fundamental. In particular, the underlying *control flow* of test-time search is often entangled with domain-specific engineering.

We argue that, for LLM-based equation discovery, compute allocation is a first-order design choice. Once an informative verifier is available, different search procedures can be understood primarily as different ways of allocating a fixed test-time budget across exploration and exploitation. This viewpoint places classical TTS methods and recent evolution-style systems in a common framework. Best-of-N (Song et al., 2025) corresponds to one-shot wide exploration, sequential refine-

* Equal Contribution. Code and data are available at https://ahong-lin.github.io/ScaleSR-tts/.

ment (Madaan et al., 2023; Chen et al., 2023) corresponds to narrow multi-step exploitation, tree search (Chen et al., 2024) corresponds to structured branching with repeated pruning, and evolution-style systems (Novikov et al., 2025; Lange et al., 2025) correspond to persistent multi-candidate or multi-island search with repeated selection and expansion. Under this view, the central object of study is not a particular agent scaffold, but the control law governing which candidates to expand, how many continuations to generate, and how aggressively to retain or discard candidates over time.

To study this question, we formulate LLM-driven equation discovery as an iterative search process with four generic operations: *sample*, *generate*, *evaluate*, and *prune*. This abstraction isolates compute allocation from domain-specific heuristics and lets us compare different control flows under a shared budget. We then instantiate the framework with two simple, highly parallel controllers: *Parallel Beam Search* (PBeam), which expands the current frontier and keeps the strongest candidates, and *Parallel Iterative Expansion* (PIE), which instead guides future expansions from the full search history. These controllers are intentionally minimal. Their purpose is not to encode human-crafted priors, but to provide a clean testbed for asking which allocation decisions matter.

Our experiments on LLM-SRBench automatic equation discovery reveal a simple and consistent picture. The dominant control variable is *search width*. Neither greedy exploitation nor one-shot broad exploration performs best. Instead, the best-performing width lies in the interior and shifts upward as the total test-time budget increases. This same allocation choice also improves wall-clock efficiency: moderate width exposes more parallelism, so better search quality need not come at the cost of slower runtime. By contrast, after width is chosen, the exact split between population size and branching factor, as well as the choice between PBeam and PIE, has a much smaller effect.

Our goal is therefore not to propose a universal controller for scientific discovery, nor to claim that equation discovery fully represents all scientific-discovery problems. Rather, we use equation discovery as a controlled benchmark setting to show that *control flow itself* is an important object of study for open-ended LLM search. Given an informative evaluator, there is substantial empirical structure in how a fixed compute budget should be spent. Making that structure explicit yields both a unified language for comparing existing equation-discovery systems and a practical recipe for scaling this class of test-time search problems.

Contributions.

- We formulate LLM-driven equation discovery as a unified external TTS process that subsumes common control flows including Best-of-N, sequential refinement, tree search, and recent evolution-style systems.
- On LLM-SRBench automated equation discovery, we show that search width is the dominant allocation parameter. In our sweep, larger budgets tend to favor wider searches, and optimizing this trade-off can improve both final performance and wall-clock efficiency.
- Our analysis is based on a large-scale controlled empirical study spanning roughly 4,000 H100 GPU hours, and we will release our code, results, prompts, and evaluation framework to support future research on TTS for equation discovery.

1.1 Background: TTS for equation discovery

Overview of TTS TTS refers to improving an LLM's performance at inference time by allocating additional computation without updating model parameters (Welleck et al., 2024; Snell et al., 2024). TTS has shown strong gains on reasoning-intensive domains, especially mathematics and code generation (Wang et al., 2023b; Brown et al., 2024; Li et al., 2025; Snell et al., 2024). In this paper, we focus on *external* TTS, where extra compute is realized through multiple model calls. We use *internal* TTS to denote the complementary regime where a single response is allowed to consume more reasoning tokens.

External TTS For clarity, we organize external TTS by control flow. *Parallel* scaling generates multiple candidates independently and aggregates them afterward, e.g., via self-consistency, repeated sampling, or verifier-based Best-of-N selection (Wang et al., 2023b; Brown et al., 2024; Cobbe et al., 2021). *Sequential* scaling iteratively refines intermediate outputs, conditioning subsequent generation steps on prior states to enable self-correction (Gou et al., 2023).*Hybrid* scaling combines branching with sequential refinement, yielding explicit search over candidate states rather than a single left-to-right rollout, thereby enabling iterative revision, look-ahead, or backtracking (Yao et al., 2023; Li et al., 2025).

Verification The performance of TTS depends critically on *verification*. In benchmark reasoning tasks, verification is often implemented by learned verifiers or reward models: outcome reward models score final answers, whereas process reward models evaluate intermediate reasoning steps (Cobbe et al., 2021; Lightman et al., 2023). These signals can be used either to guide search online or to rerank candidate solutions offline.

TTS in scientific discovery Recent work has used an LLM to propose candidate programs or constructions, while an external evaluator (e.g., mathematical objective) assigns fitness and closes the search loop (Romera-Paredes et al., 2024; Novikov et al., 2025). This discovery setting differs from standard math or coding benchmarks in an important respect: the objective is typically *open-ended*. There is often no unique ground-truth answer known in advance; instead, progress is measured by improvement under a task-specific evaluator. When that evaluator is reliable, the main bottleneck shifts from recovering a known answer to allocating test-time compute effectively over a large search space of candidate solutions. Under this lens, recent self-evolving systems (Lange et al., 2025; Assumpção et al., 2025) can be viewed as specialized instances of external TTS that differ primarily in their control flow.

1.2 A unified TTS formulation

We formalize LLM-driven equation discovery as an iterative search over a dynamic memory state $\mathcal{M}_t$, which maintains the candidate solutions, such as equations or executable programs, available at step t. $\mathcal{M}_0$ can be initialized as a naive solution to the problem. By abstracting away domain-specific details such as prompt design or mutation rules, we obtain a common computational framework governed by a global test-time compute budget N. Each iteration is specified by four ingredients: a selection policy over $\mathcal{M}_t$, a branching rule, a verifier, and a pruning operator.

Sample Let $q_t(\cdot \mid \mathcal{M}_t)$ denote a selection policy over the current memory state. Select a subset

$$S_t = \{x_t^{(i)}\}_{i=1}^{n_t} \sim q_t(\cdot \mid \mathcal{M}_t),$$
$$S_t \subseteq \mathcal{M}_t, \qquad |S_t| = n_t.$$

for expansion. The policy q_t may be greedy or score-based, and determines which previously discovered candidates are revisited at step t.

Generate For each candidate $x_t^{(i)} \in S_t$,[1] query the LLM to generate a set $G_t^{(i)}$ of $k_t^{(i)}$ new proposals (e.g., refinements or novel variations). The full generated set is

$$G_t = \bigcup_{i=1}^{n_t} G_t^{(i)}, \qquad |G_t| = \sum_{i=1}^{n_t} k_t^{(i)}.$$

When $k_t^{(i)} = k_t$ for all i, we write k_t for the common branching factor.

Evaluate and prune Score G_t using a task-specific verifier, merge the new proposals with the current memory, and apply a pruning operator Π_t (e.g., FIFO, top-B selection or diversity filtering) to retain the surviving candidates:

$$\mathcal{M}_{t+1} = \Pi_t(\mathcal{M}_t \cup G_t).$$

Iterate Repeat this process for T steps until the global compute budget N is exhausted. Assuming a uniform cost per generated proposal,[2] the total test-time compute constraint is

$$\sum_{t=0}^{T-1} \sum_{i=1}^{n_t} k_t^{(i)} \le N.$$

This framework can express most external TTS control flows: *parallel scaling* maximizes the branching factor in a single step ($T = 1$), *sequential scaling* iteratively refines candidates with minimal branching, and *hybrid scaling* alternates between selection, branching, and pruning over multiple rounds. Under this view, disparate discovery algorithms differ primarily in their parameterization of

$$\big(q_t,\ n_t,\ \{k_t^{(i)}\}_{i=1}^{n_t},\ \Pi_t\big);$$

in the common constant-branching case, this reduces to (q_t, n_t, k_t, Π_t). Crucially, this abstraction isolates our core research question: assuming access to an informative verifier, how should test-time compute be allocated across selection, branching, and pruning to maximize equation-discovery progress under a fixed budget?

[1] In practice, we can select multiple candidates to construct the context for the LLM. This results in a directed acyclic graph rather than a tree in the control flow, although the parameterization remains similar. For simplicity, we follow recent practices (Wang et al., 2025b; Yuksekgonul et al., 2026) to focus on the single candidate setting.

[2] Traditional TTS literature use FLOPs as their compute measure, which is mainly affected by the number of model rollouts (i.e., N) if we only focus on the same model for the same problem. The justification is detailed in section A.

Simplified notation. In the remainder of the paper, for simplicity, we focus on time-homogeneous controllers with $n_t \equiv n$, $k_t^{(i)} \equiv k$, and a fixed pruning rule $\Pi_t \equiv \Pi$, and write the controller as (q, n, k, Π). We further define the per-iteration search width as $w = nk$.

2 Method

2.1 Diagnosis: Are classical control flows suitable for equation discovery?

Desiderata: performance and efficiency To evaluate the suitability of existing control flows for equation discovery, we establish two primary desiderata: *performance* and *efficiency*. Under a fixed compute budget N, **performance** relies on balancing exploration, which searches broadly over possible functional forms, and exploitation, which refines promising candidates. In our unified framework, this requires a strategic parameterization of the controller; in the simplified setting studied below, this reduces to the selection rule q, the number of expanded candidates n, the branching factor k, and the pruning operator Π. **Efficiency**, conversely, is governed by wall-clock time and verification costs. In equation-discovery settings, the verifier may involve fitting parameters, running numerical evaluation, or executing candidate programs. Consequently, higher efficiency directly yields better performance under a fixed time budget. To accelerate wall-clock time, a practical control flow should expose sufficient parallelism by submitting large batches of evaluation jobs simultaneously. When verification is expensive, a useful control flow must carefully tune (q, n, k, Π) to push the performance Pareto frontier under a limited evaluation budget.

Evaluating classical baselines How do classical algorithms meet these desiderata in equation discovery? As intuitively illustrated in Figure 1, we evaluate three dominant paradigms through the lens of our (q, n, k, Π) abstraction:

- **Extreme Scaling (e.g., Self-Consistency, Self-Refine):** Methods optimizing for pure parallelism ($T = 1$, maximal k) or pure sequentiality ($n = 1$, minimal k) fail to balance exploration and exploitation, making them poorly suited for the large search spaces that characterize automated equation discovery.
- **Tree Search:** While gracefully balancing exploration and exploitation (e.g., MCTS), sequential node-expansion and backpropagation inherently restrict n and k at any given step. When verifiers are slow, this strict sequentiality creates wall-clock bottlenecks that can reduce overall search efficiency.
- **Evolutionary Systems**: Recent self-evolving frameworks (e.g., OpenEvolve (Sharma, 2025)) balance exploration and efficiency by maintaining multiple isolated populations ("islands") and expanding them sequentially in parallel ($n > 1$). However, these systems are often engineered with complex features such as LLM ensembles, explicit crossover rules, and specialized prompting heuristics. It remains unclear which of these features are necessary for equation discovery.

We hypothesize that much of the empirical benefit of evolutionary methods can be explained by a simpler mechanism: improved compute allocation. In our unified (q, n, k, Π) view, this suggests a reduced design space centered on a small number of allocation choices: how many candidates are expanded per round, how many continuations each candidate spawns, and how much the search is partitioned into semi-independent groups. This reframing lets us compare heavily engineered systems to minimal controllers and ask directly which allocation decisions matter for performance and efficiency in this setting.

2.2 Simple yet expressive TTS control flows

Motivated by this hypothesis, we instantiate a simple family of highly parallel controllers with minimal human-crafted features. We parameterize the search by three practical knobs: population size n, per-parent branching factor k, and the number of groups g. We define the *global width* as $w = nk$, i.e., the total number of expansions produced in one synchronized iteration. The role of g is inspired by island models in evolutionary systems (Novikov et al., 2025), but we treat it more generally as a partitioning variable that changes both search isolation and synchronization granularity. Increasing g creates more independent sub-searches and smaller synchronization units; whether this improves diversity is an empirical question. This reduced (n, k, g) design space is therefore expressive enough to recover useful hybrid control flows while remaining clean enough for systematic study.

To study this reduced design space without confounding domain-specific heuristics, we instantiate two clean, parallel control flows that differ mainly

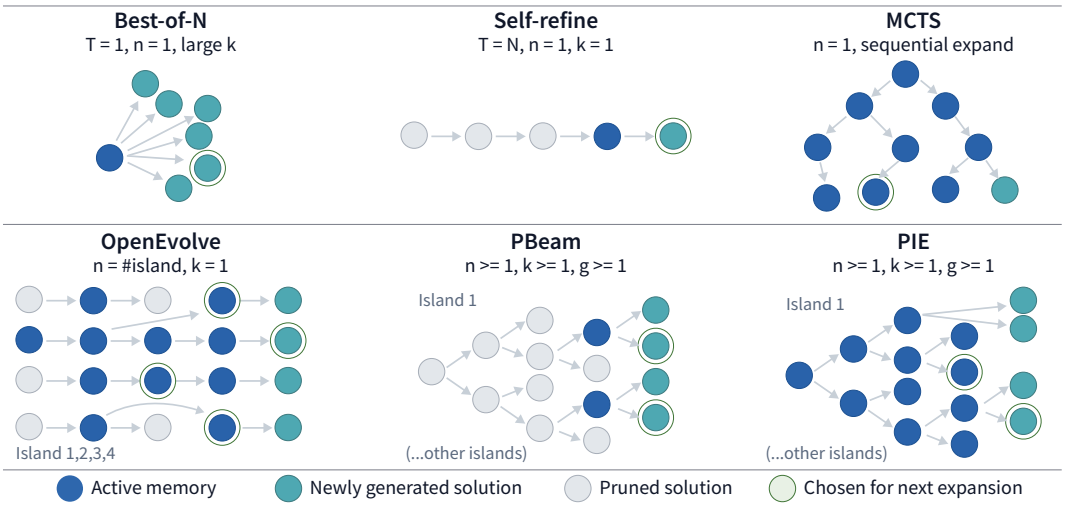

Figure 1: **Comparison of TTS control flows under the unified formulation.** Top row: classical baselines emphasize only one aspect of compute allocation: Best-of-N performs purely parallel one-shot sampling ($T = 1$, $n = 1$, k large), Self-refine performs purely sequential single-chain refinement ($T = N$, $n = 1$, $k = 1$), and MCTS balances exploration and exploitation through sequential tree expansion but with limited parallelism. Bottom row: hybrid, equation-discovery-oriented methods maintain multiple active candidates and combine branching with pruning. OpenEvolve expands multiple isolated islands in parallel ($n = \#\text{islands}$, $k = 1$); our Parallel Beam Search (PBeam) performs beam-style expansion within each island ($n \geq 1, k \geq 1, g \geq 1$); and Parallel Iterative Expansion (PIE) further relaxes strict beam pruning by expanding from the full historical pool.

in their selection-and-pruning policies while sharing the same allocation variables (n, k, g):

Parallel Beam Search (PBeam): We partition the active population and the total budget evenly across g independent groups. Unless otherwise stated, n, k, and $w = nk$ denote *global* quantities, so each group operates with local population n/g, local width $w/g = (n/g)k$, and local budget N/g. Omitting the group index for clarity, within each group we execute a constant-width beam search. At step t, the selection policy chooses the entire current frontier,

$$S_t = \mathcal{M}_t, \qquad |S_t| = |\mathcal{M}_t| = n/g.$$

The model generates k candidates per selected candidate, yielding a generated set of size

$$|G_t| = (n/g)k = w/g.$$

The pruning operator Π strictly retains the top-n/g candidates according to the verifier score $f(x)$:

$$\mathcal{M}_{t+1} = \mathrm{Top}_{n/g}(\mathcal{M}_t \cup G_t; f),$$

where $\mathrm{Top}_m(A; f)$ denotes the m elements of A with the largest values of f.

Parallel Iterative Expansion (PIE): Building upon PBeam, PIE relaxes the strict frontier-based selection rule within each group to better preserve exploration. Instead of truncating unselected candidates, we maintain the full generation history

$$\mathcal{H}_{t+1} = \mathcal{H}_t \cup G_t, \qquad \mathcal{H}_0 = \mathcal{M}_0.$$

We then use a softmax selection policy over historical scores to choose n/g candidates for the next expansion step. The probability of selecting a candidate $x \in \mathcal{H}_{t+1}$ is

$$p_{t+1}(x) = \frac{\exp(f(x)/\tau)}{\sum_{x' \in \mathcal{H}_{t+1}} \exp(f(x')/\tau)},$$

where τ is a temperature hyperparameter controlling the exploration–exploitation tradeoff. The pruning operator simply updates the memory state:

$$\mathcal{M}_{t+1} = \mathcal{H}_{t+1}.$$

These algorithms instantiate two simple choices of selection-and-pruning policy within the broader (q_t, n_t, k_t, Π_t) framework, while keeping the tunable allocation space small and highly parallel. We restrict our current experiments to these two control flows, leaving to future work the optimization of (1) richer selection policies for choosing S_t (e.g., PUCT); (2) adaptive branching methods to dynamically determine $k_t^{(i)}$ for each node $x_t^{(i)}$; (3) diversity-aware pruning operators to maintain population quality during extended rollouts; and (4) iteration-aware allocation for adjusting expansion strategies for different t (e.g., ASHA).

3 Experiments

3.1 Experimental setting

Benchmarks and models We evaluate on LLM-SRBench, a benchmark for scientific equation discovery (Shojaee et al., 2025). We focus on the Bio and Material splits, which contain 24 and 25 tasks, respectively. Each task provides a natural-language problem description, variable definitions, a training set, and a held-out test set. Given the problem description and training data, the model proposes a Python program that specifies both the functional form of the equation and an optimizer for fitting its free parameters. During test-time search, the model may observe verifier feedback (fitness) on the training split of previous proposals, but it never accesses the test set. For each random seed, we first average task-level results over all tasks in a domain, and then report the mean and standard deviation over three seeds. Our main study focuses on `gpt-oss-20b` (OpenAI, 2025) on Bio. We use `gpt-oss-20b` on Material as a dataset-generalization check, and `Qwen3-30B-A3B` (Yang et al., 2025) on Bio as a model-generalization check. We provide more details (e.g., task prompts, dataset statistics, initial solutions) in section B for readers to understand this benchmark.

Search space We sweep PBeam and PIE under total budgets $N \in \{1, 2, 4, 8, 16, 32, 64, 128\}$. We vary the population size n, per-node branching factor k, group size g, and number of iterations T, with $N = (n \times k) \times T = w \times T$. Grouping partitions the active population into g independent groups, so the effective width of each group is w/g. To keep the sweep manageable, we first fix $g = 1$ and sweep $(n, k, T) = (2^i, 2^j, 2^k)$, where $i + j + k \in [0, 7]$ and $i, j, k \in \mathbb{N}$. We then sweep g for the competitive (n, k, T) settings identified in the first stage. Detailed sweep grids, hyperparameters, and results are deferred to section B.4.

Metrics We follow LLM-SRBench to use $\mathrm{Acc}_{0.1}$ as the primary metric:

$$\mathrm{Acc}_{0.1} = \mathbb{1}\left(\max_{1 \le i \le m} \left| \frac{\hat{y}_i - y_i}{y_i} \right| \le 0.1\right),$$

In compute allocation study, we report training $\mathrm{Acc}_{0.1}$ as our primary metric. We defer the interpretation of train–test discrepancies section 5, because our main goal here is to understand compute allocation under a fixed verifier rather than to optimize the verifier itself. We do not report other fitness metrics like NMSE and R^2 due to large discrepancy in scales between different tasks, which makes them hard to aggregate.

3.2 Result analysis

Search width dictates both performance and wall-clock efficiency. We plot the performance envelope over PIE and PBeam for each width and budget in Figure 2. Across domains (Bio, Material) and models (`gpt-oss-20b`, `Qwen3-30B-A3B`), search width is the most important allocation variable. Very narrow searches over-emphasize sequential refinement, while one-shot wide searches underuse iterative feedback. The best-performing configurations in our sweep usually lie between these extremes.

We deliberately treat the budget–width relationship as an empirical trend rather than a universal scaling law. The sweep contains only a small number of discrete budget levels, and several nearby widths often perform similarly. Nevertheless, the qualitative pattern is consistent: small budgets favor narrow searches, whereas larger budgets tend to benefit from moderately wider synchronized expansion.

Crucially, increasing width can also improve wall-clock efficiency. Larger widths increase model inference parallelism and reduce the number of sequential refinement rounds ($T = N/w$), so the wall-clock Pareto frontier favors moderate-to-large widths (Figure 3). For example, on the Bio dataset, shifting from greedy refinement ($w = 1$) to width 32 reduces runtime by nearly 80% (from 7622 s to 1586 s) while simultaneously improving train $\mathrm{Acc}_{0.1}$ from 0.965 to 0.984.

Grouping improves efficiency via reduced synchronization. While width dictates the primary time–quality trade-off, grouping acts mainly as a secondary systems knob. Holding the total budget N and the global width w fixed, increasing the number of groups g yields a modest wall-clock speedup because synchronization occurs within groups of size w/g rather than across all w expansions. This reduces idle time from stragglers and improves hardware utilization when model or verifier latency is heterogeneous. Empirically, moving from $g = 1$ to $g = 8$ yields up to a $1.075\times$ speedup. By contrast, the effect on search quality is weak and not consistently positive: in our equation-discovery benchmark, island-style isolation does not provide a reliable diversity benefit once width

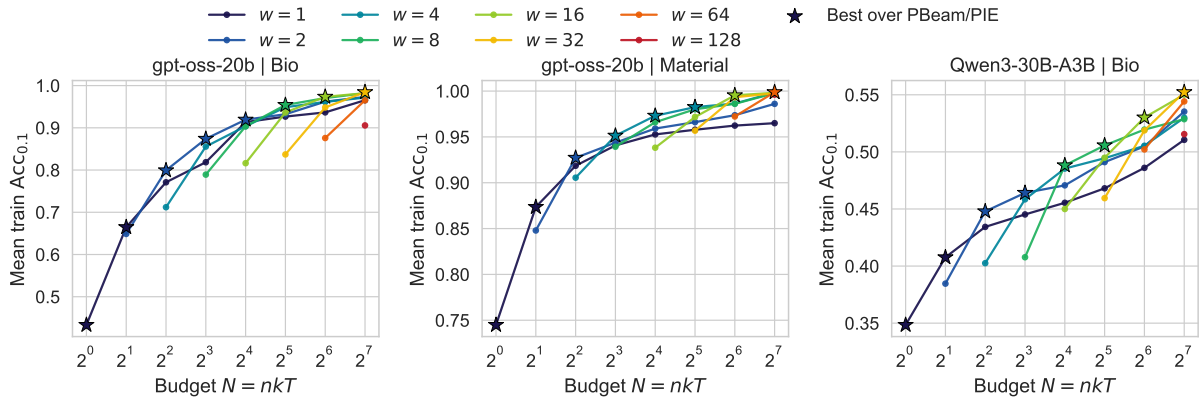

Figure 2: **Search-width frontiers across domains and backbones.** Each curve corresponds to a fixed width $w = nk$ ($g = 1$), with colors transitioning from dark (small w) to bright (large w). Stars mark the best attainable point at each budget over both PBeam and PIE. Across the budgets considered, the envelope tends to shift from narrower to wider searches, indicating that width is a key allocation variable.

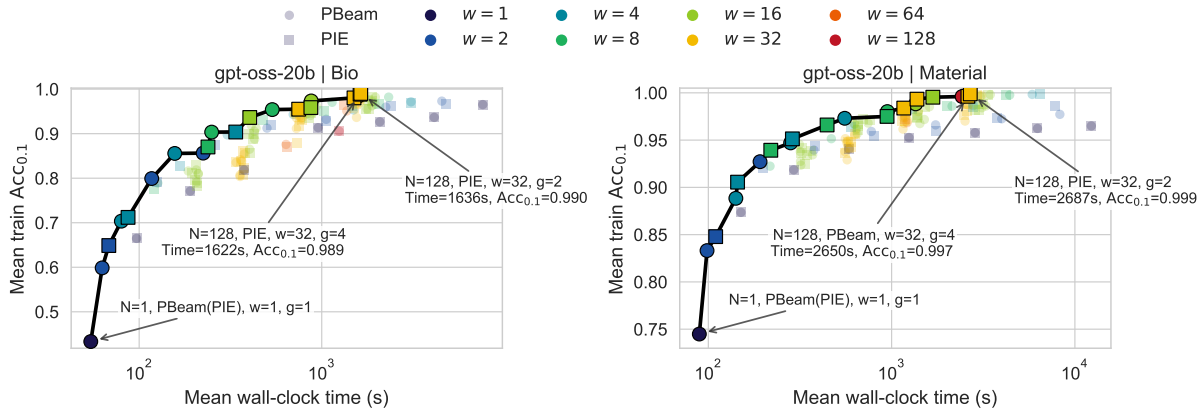

Figure 3: **Wall-clock frontier under different compute-allocation strategies.** Each point is the best measured configuration at fixed $(N, w, g, \text{algorithm})$ after optimizing over the remaining decomposition. Marker shape indicates the algorithm, color indicates width, and the black curve shows the time–quality Pareto frontier.

is already chosen. Over-grouping can even hurt, because it reduces both the per-group width w/g and the per-group budget N/g, pushing each group toward a narrower search regime than the one preferred by the global budget (detailed discussion in section C.3). We therefore treat g as a throughput parameter rather than a search-quality parameter.

Other choices are second-order effects. Once the global width $w = nk$ is fixed, the remaining design choices have a minor effect on performance. First, the exact decomposition of w into population size n and branching factor k matters much less than choosing the right width in the first place (table 6). Second, the difference between PBeam and PIE is also small: their train $\text{Acc}_{0.1}$ gap usually stays within $\pm 2\%$ across budgets and widths (fig. 4). Both effects are substantially smaller than the cross-width differences reported above. In this sense, width determines the operating regime, while the exact n–k split and the choice between PBeam and PIE mainly fine-tune performance. We defer the detailed ablations to section C.2.

An empirical recipe for workflow design. Synthesizing these findings, we recommend the following top-down strategy for deploying external test-time search. First, maximize the compute budget by selecting the largest budget N permitted by your inference hardware and latency constraints. Next, determine the search width w using a small pilot sweep over powers of two, or use the empirical width frontier in Section C.1 as a coarse warm start. Once w has been chosen, set the remaining hyperparameters to moderate values, using balanced choices such as $n/w \geq 1/8$, $k \in [2, 8]$, and a mild g. Finally, treat the choice among selection rules (e.g., PIE, PBeam, or more complex systems) as a minor refinement step.

4 Related work

LLM-driven scientific discovery Recent advances in LLMs have enabled them to augment, accelerate, and in some cases automate

scientific discovery across the full research pipeline (AI4Science and Quantum, 2023; Wang et al., 2023a). Scientific discovery is often viewed as two tightly coupled stages: hypothesis generation and empirical verification. Prior work shows that LLMs can act as effective hypothesis generators by drawing on broad domain knowledge and reasoning ability (Qi et al., 2023; Si et al., 2024). When combined with external tools and agentic workflows, they can also execute experiments, analyze data, and iteratively refine candidate hypotheses (Ma et al., 2024; Majumder et al., 2024). This paradigm has shown promise in domains such as chemistry (Wang et al., 2025a), biomedicine (Gao et al., 2024), and astronomy (Sun et al., 2025).

Equation discovery Equation discovery, or symbolic regression (SR), is a long-standing problem in scientific modeling that seeks equations balancing predictive accuracy and simplicity for a given dataset (Makke and Chawla, 2024). Classical SR methods mainly rely on search-based optimization (Koza, 1994; Virgolin et al., 2021; Cranmer, 2023). Other approaches use Monte Carlo search (Jin et al., 2019; Sun et al., 2022) or reinforcement learning (Petersen et al., 2019; Landajuela et al., 2021; Mundhenk et al., 2021). More recent work has explored pretrained Transformer-based SR models (Biggio et al., 2021; Valipour et al., 2021; Kamienny et al., 2022; Vastl et al., 2024). LLM-based SR uses LLMs to generate equation hypotheses and interact with TTS workflows to iteratively improve them while leveraging knowledge encoded in the model (Shojaee et al., 2024; Meyerson et al., 2024; Grayeli et al., 2024).

5 Extended Discussion

Simple control flows beat complicated systems. We also compare the lightweight methods PBeam and PIE with OpenEvolve, as an example of a more engineered evolutionary system. The results show that, when width and depth are controlled to be the same, PBeam and PIE remain competitive. When their parameters are set according to our empirical recipe (section 3.2), PBeam and PIE perform slightly better than OpenEvolve, partly because OpenEvolve does not allow $k > 1$ or mild grouping with $g < w$, making it less expressive. Detailed slice-level comparisons are reported in fig. 5. However, this does not imply that crossover, ensembling, or specialized prompting are never useful. Rather, on this benchmark, their contribution appears secondary to getting the global compute allocation right. An implication: new discovery agents should be compared against width-matched minimal baselines; otherwise, gains from additional engineering may be confounded with gains from simply operating in a better width–depth regime.

Connecting to other TTS literature. Our work is complementary to Wu et al. (2024), who study compute-optimal inference for closed-ended math reasoning. Under a fixed budget, they ask which combination of model size and existing control flow (tree search vs. sampling) maximizes answer accuracy. In contrast, our setting is open-ended scientific discovery with an external verifier, where the central problem is not only *which* model or *which* control flow to use, but *how* to allocate a fixed budget under a *unified* control flow.

Verifier quality is the other bottleneck. Our conclusions should be read as conditional on the verifier being informative enough to rank candidates meaningfully. In conventional TTS, learned reward models can become the limiting factor because additional search amplifies reward misspecification. Scientific discovery changes the source of the verifier, but not this principle: even when feedback comes from experimental proxies, search can still overfit to the measured objective rather than the true scientific target. In our benchmark, the verifier is based on training-set performance, so the search is optimized against an imperfect proxy (discussed in section C.4, though the train-test gap is not severe for this benchmark). This suggests a useful decomposition: better control flow improves how efficiently the system explores under a fixed verifier, whereas better verification improves what the system is ultimately driven toward. Both matter, and improvements in verifier fidelity may also shift the optimal exploration–exploitation balance.

6 Conclusion

We study test-time scaling for LLM-based equation discovery through the lens of compute allocation. On LLM-SRBench, search width is the dominant factor in our sweep: larger budgets generally favor wider searches, and appropriate width selection improves both accuracy and wall-clock efficiency. These results highlight compute allocation as a central design choice for equation discovery with informative verifiers, and motivate its study in broader open-ended scientific search settings.

Limitations

This work has several limitations. First, our empirical conclusions are specific to equation discovery as instantiated by LLM-SRBench, especially the Biology and Material Science splits considered in our experiments. Although equation discovery is a useful controlled setting for studying open-ended search with an external verifier, it does not capture the full complexity of scientific discovery, such as experimental design, noisy measurement processes, causal interpretation, safety constraints, or long-horizon laboratory validation. Therefore, the observed width–budget trade-off should be interpreted as evidence for this benchmark setting rather than as a universal law for all scientific-discovery workflows.

Second, our analysis assumes access to an informative verifier. In our experiments, search is guided by training-set performance, and we use training $\mathrm{Acc}_{0.1}$ as the main metric for studying compute allocation. This design helps isolate the effect of test-time compute allocation under a fixed verifier, but it also means that stronger search can overfit to the training split or to the specific proxy objective used by the benchmark. Although the train–test gap is not severe in our reported setting, improved verifier fidelity remains an important bottleneck for applying these methods to real scientific problems.

Third, our study focuses on a limited set of models, domains, and controller families. We evaluate primarily with `gpt-oss-20b`, include `Qwen3-30B-A3B` as an auxiliary cross-backbone check, and instantiate two minimal parallel controllers, PBeam and PIE. These choices are sufficient for isolating the role of width, branching, grouping, and depth in a controlled sweep, but they do not cover the full space of possible language models, prompting strategies, learned selection policies, diversity mechanisms, adaptive branching rules, or richer evolutionary systems. As a result, the exact optimal width and controller behavior may change with different backbones, prompts, verifiers, or task families.

Fourth, our compute measure abstracts test-time cost by the number of generated proposals N. This is appropriate for comparing allocation strategies under a fixed model and prompt template, but it does not fully capture all practical costs. In real deployments, token lengths, model-serving latency, verifier runtime, hardware utilization, failed executions, and synchronization overhead can vary across tasks and candidate programs. Our wall-clock analysis partially addresses this issue, but a more complete systems-level accounting would be needed before transferring these findings directly to heterogeneous production or laboratory settings.

Finally, our work studies how to allocate inference-time compute more effectively; it does not by itself guarantee scientifically valid discoveries. The generated equations are candidate models optimized against benchmark feedback, and strong benchmark performance should not be interpreted as a substitute for domain expertise, independent validation, or experimental confirmation. This limitation is especially important for high-stakes scientific or engineering applications, where incorrect but plausible equations could lead to misleading conclusions.

Ethical Considerations

This work studies test-time compute allocation for scientific equation discovery in a benchmark setting. Our goal is to understand how to allocate inference-time compute more effectively, rather than to automate high-stakes scientific decision-making without oversight.

The main risk of systems of this kind is that they can generate plausible but incorrect equations, hypotheses, or programs, especially when the verifier is imperfect or only measures a proxy objective. In our setting, the search is optimized against benchmark feedback on the training split, so strong search performance should not be interpreted as a validated scientific discovery. Any real-world use would require domain-expert review and additional experimental validation.

A second consideration is computational cost. Large-scale test-time search can consume substantial compute resources. One motivation of this work is therefore efficiency: we study how better compute allocation can improve performance while also reducing wall-clock time and synchronization overhead.

Finally, we plan to release code, prompts, and evaluation scripts for the benchmark setting used in this paper to support reproducibility. However, we do not claim that improved benchmark performance alone implies safe or reliable deployment in real scientific practice.

References

Microsoft Research AI4Science and Microsoft Azure Quantum. 2023. The impact of large language models on scientific discovery: a preliminary study using gpt-4. *arXiv preprint arXiv:2311.07361*.

Henrique Assumpção, Diego Ferreira, Leandro Campos, and Fabricio Murai. 2025. Codeevolve: An open source evolutionary coding agent for algorithm discovery and optimization. *arXiv preprint arXiv:2510.14150*.

Luca Biggio, Tommaso Bendinelli, Alexander Neitz, Aurelien Lucchi, and Giambattista Parascandolo. 2021. Neural symbolic regression that scales. In *International conference on machine learning*, pages 936–945. Pmlr.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. 2024. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Ziru Chen, Michael White, Ray Mooney, Ali Payani, Yu Su, and Huan Sun. 2024. When is tree search useful for llm planning? it depends on the discriminator. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13659–13678.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Miles Cranmer. 2023. Interpretable machine learning for science with pysr and symbolicregression. jl. *arXiv preprint arXiv:2305.01582*.

Shanghua Gao, Ada Fang, Yepeng Huang, Valentina Giunchiglia, Ayush Noori, Jonathan Richard Schwarz, Yasha Ektefaie, Jovana Kondic, and Marinka Zitnik. 2024. Empowering biomedical discovery with AI agents. *Cell*, 187(22):6125–6151.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2023. Critic: Large language models can self-correct with tool-interactive critiquing. *arXiv preprint arXiv:2305.11738*.

Arya Grayeli, Atharva Sehgal, Omar Costilla-Reyes, Miles Cranmer, and Swarat Chaudhuri. 2024. Symbolic regression with a learned concept library. *Advances in Neural Information Processing Systems*, 37:44678–44709.

Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. 2025. Rstar-math: Small llms can master math reasoning with self-evolved deep thinking. *arXiv preprint arXiv:2501.04519*.

Ying Jin, Weilin Fu, Jian Kang, Jiadong Guo, and Jian Guo. 2019. Bayesian symbolic regression. *arXiv preprint arXiv:1910.08892*.

Pierre-Alexandre Kamienny, Stéphane d'Ascoli, Guillaume Lample, and François Charton. 2022. End-to-end symbolic regression with transformers. *Advances in Neural Information Processing Systems*, 35:10269–10281.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

John R. Koza. 1994. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2):87–112.

Mikel Landajuela, Brenden K Petersen, Sookyung Kim, Claudio P Santiago, Ruben Glatt, Nathan Mundhenk, Jacob F Pettit, and Daniel Faissol. 2021. Discovering symbolic policies with deep reinforcement learning. In *International Conference on Machine Learning*, pages 5979–5989. PMLR.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. 2025. Shinkaevolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*.

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E. Gonzalez, and Ion Stoica. 2025. S*: Test time scaling for code generation. *arXiv preprint arXiv:2502.14382*.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. *arXiv preprint arXiv:2305.20050*.

Pingchuan Ma, Tsun-Hsuan Wang, Minghao Guo, Zhiqing Sun, Joshua B Tenenbaum, Daniela Rus, Chuang Gan, and Wojciech Matusik. 2024. Llm and simulation as bilevel optimizers: A new paradigm to advance physical scientific discovery. *arXiv preprint arXiv:2405.09783*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594.

Bodhisattwa Prasad Majumder, Harshit Surana, Dhruv Agarwal, Sanchaita Hazra, Ashish Sabharwal, and Peter Clark. 2024. Data-driven discovery with large generative models. *arXiv preprint arXiv:2402.13610*.

Nour Makke and Sanjay Chawla. 2024. Interpretable scientific discovery with symbolic regression: a review. *Artificial Intelligence Review*, 57(1):2.

Elliot Meyerson, Mark J Nelson, Herbie Bradley, Adam Gaier, Arash Moradi, Amy K Hoover, and Joel Lehman. 2024. Language model crossover: Variation through few-shot prompting. *ACM Transactions on Evolutionary Learning*, 4(4):1–40.

T. Nathan Mundhenk, Mikel Landajuela, Ruben Glatt, Claudio P. Santiago, Daniel M. Faissol, and Brenden K. Petersen. 2021. Symbolic regression via neural-guided genetic programming population seeding. In *Advances in Neural Information Processing Systems*.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. 2025. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*.

OpenAI. 2025. gpt-oss-120b & gpt-oss-20b model card. https://openai.com/index/gpt-oss-model-card/. Published August 5, 2025.

Brenden K Petersen, Mikel Landajuela, T Nathan Mundhenk, Claudio P Santiago, Soo K Kim, and Joanne T Kim. 2019. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. *arXiv preprint arXiv:1912.04871*.

Biqing Qi, Kaiyan Zhang, Haoxiang Li, Kai Tian, Sihang Zeng, Zhang-Ren Chen, and Bowen Zhou. 2023. Large language models are zero shot hypothesis proposers. *arXiv preprint arXiv:2311.05965*.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. 2024. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475.

Asankhaya Sharma. 2025. Openevolve: an open-source evolutionary coding agent.

Parshin Shojaee, Kazem Meidani, Shashank Gupta, Amir Barati Farimani, and Chandan K Reddy. 2024. Llm-sr: Scientific equation discovery via programming with large language models. *arXiv preprint arXiv:2404.18400*.

Parshin Shojaee, Ngoc-Hieu Nguyen, Kazem Meidani, Amir Barati Farimani, Khoa D. Doan, and Chandan K. Reddy. 2025. LLM-SRBench: A new benchmark for scientific equation discovery with large language models. *arXiv preprint arXiv:2504.10415*.

Chenglei Si, Diyi Yang, and Tatsunori Hashimoto. 2024. Can llms generate novel research ideas? a large-scale human study with 100+ nlp researchers. *arXiv preprint arXiv:2409.04109*.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*.

Yifan Song, Guoyin Wang, Sujian Li, and Bill Yuchen Lin. 2025. The good, the bad, and the greedy: Evaluation of llms should not ignore non-determinism. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 4195–4206.

Fangzheng Sun, Yang Liu, Jian-Xun Wang, and Hao Sun. 2022. Symbolic physics learner: Discovering governing equations via monte carlo tree search. *arXiv preprint arXiv:2205.13134*.

Zechang Sun, Yuan-Sen Ting, Yaobo Liang, Nan Duan, Song Huang, and Zheng Cai. 2025. Mephisto: Self-Improving Large Language Model-Based Agents for Automated Interpretation of Multi-band Galaxy Observations. *arXiv e-prints*, arXiv:2510.08354.

Mojtaba Valipour, Bowen You, Maysum Panju, and Ali Ghodsi. 2021. Symbolicgpt: A generative transformer model for symbolic regression. *arXiv preprint arXiv:2106.14131*.

Martin Vastl, Jonáš Kulhánek, Jiří Kubalík, Erik Derner, and Robert Babuška. 2024. Symformer: End-to-end symbolic regression using transformer-based architecture. *IEEE Access*, 12:37840–37849.

Marco Virgolin, Tanja Alderliesten, Cees Witteveen, and Peter AN Bosman. 2021. Improving model-based genetic programming for symbolic regression of small expressions. *Evolutionary computation*, 29(2):211–237.

Hanchen Wang, Tianfan Fu, Yuanqi Du, Wenhao Gao, Kexin Huang, Ziming Liu, Payal Chandak, Shengchao Liu, Peter Van Katwyk, Andreea Deac, Anima Anandkumar, Karianne Bergen, Carla P. Gomes, Shirley Ho, Pushmeet Kohli, Joan Lasenby, Jure Leskovec, Tie-Yan Liu, Arjun Manrai, and 11 others. 2023a. Scientific discovery in the age of artificial intelligence. *Nature*, 620(7972):47–60.

Haorui Wang, Jeff Guo, Lingkai Kong, Rampi Ramprasad, Philippe Schwaller, Yuanqi Du, and Chao Zhang. 2025a. Llm-augmented chemical synthesis and design decision programs. *Forty-Second International Conference on Machine Learning*.

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

Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, Xinyu Yang, Zeyi Huang, Xuehai He, Luyao Ma, Baolin Peng, and 1 others. 2025b. Thetaevolve: Test-time learning on open problems. *arXiv preprint arXiv:2511.23473*.

Sean Welleck, Amanda Bertsch, Matthew Finlayson, Hailey Schoelkopf, Alex Xie, Graham Neubig, Ilia Kulikov, and Zaïd Harchaoui. 2024. From decoding to meta-generation: Inference-time algorithms for large language models. *Transactions on Machine Learning Research*.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. 2024. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models. *arXiv preprint arXiv:2408.00724*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 4 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

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

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, and 1 others. 2026. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*.

A Compute budget N and FLOPs

Our unified formulation measures test-time compute by the total number of generation passes,

$$N = \sum_{t=0}^{T-1} \sum_{i=1}^{n_t} k_t^{(i)},$$

namely, the total number of candidate proposals produced across all steps. While prior test-time scaling work often reports inference compute in FLOPs, under our experimental setting this choice is equivalent to measuring compute by N.

To see this, consider a single generation pass that produces one proposal. Let its input and output lengths be L_{in} and L_{out}, respectively. Following standard scaling-law analyses (Kaplan et al., 2020), the FLOPs required for one autoregressive rollout of a transformer with P parameters is approximately

$$C_{\text{rollout}} \approx 2P\big(L_{\text{in}} + L_{\text{out}}\big).$$

Now index all generated proposals in the entire search process by (t, i, j), where t is the iteration, i indexes the selected parent candidate, and $j \in \{1, \ldots, k_t^{(i)}\}$ indexes the newly generated proposal. The total inference cost is then

$$C_{\text{total}} \approx \sum_{t=0}^{T-1} \sum_{i=1}^{n_t} \sum_{j=1}^{k_t^{(i)}} 2P\big(L_{\text{in}}^{(t,i,j)} + L_{\text{out}}^{(t,i,j)}\big).$$

Under our controlled setting, the base model is fixed, so P is constant. Moreover, we study compute allocation for the same task instance: each rollout takes as context a previous program (or equation) and asks the model to produce a refined or mutated proposal of comparable size. Hence both the input and output lengths stay within a task-dependent bounded range. That is, there exist constants

$$\underline{L}_{\text{in}},\ \overline{L}_{\text{in}},\ \underline{L}_{\text{out}},\ \overline{L}_{\text{out}}$$

such that for every rollout,

$$\underline{L}_{\text{in}} \le L_{\text{in}}^{(t,i,j)} \le \overline{L}_{\text{in}}, \qquad \underline{L}_{\text{out}} \le L_{\text{out}}^{(t,i,j)} \le \overline{L}_{\text{out}}.$$

Therefore, each rollout has cost bounded between two constants:

$$c_{\min} \le C_{\text{rollout}}^{(t,i,j)} \le c_{\max},$$

where

$$c_{\min} = 2P\big(\underline{L}_{\text{in}} + \underline{L}_{\text{out}}\big), \qquad c_{\max} = 2P\big(\overline{L}_{\text{in}} + \overline{L}_{\text{out}}\big).$$

Summing over all rollouts gives

$$c_{\min} N \;\le\; C_{\text{total}} \;\le\; c_{\max} N.$$

Hence, for a fixed model on a fixed task, the total test-time FLOPs grow linearly with N. In practice, $c_{\min}$ is also close to $c_{\max}$ as each input and output consists of the same prompt template and a solution. In other words, N and FLOPs are equivalent up to a task-dependent constant factor, and therefore induce the same notion of compute budget to compare different compute-allocation strategies.

Finally, our budget N accounts only for the LLM generation cost. Auxiliary operations such as selection, pruning, and verifier evaluation are either negligible relative to model inference or held fixed across methods and thus do not affect the equivalence above.

B Benchmark and implementation details

B.1 Benchmark instantiation

Our experiments instantiate the LSR-Synth portion of LLM-SRBench and restrict attention to the two domains studied in the main paper: Biology and Material Science. At the benchmark level, these correspond to 24 biology tasks and 25 material-science tasks, yielding 49 tasks in total. In our local benchmark layout, each task directory contains four components: (i) a natural-language task specification, (ii) a shared executable scaffold, (iii) a task-specific evaluator, and (iv) a local dataset partitioned into `train`, `test`, and `ood_test` splits. The search procedure is given access only to the task specification and the training split. The held-out `test` split is queried exclusively for logging best-so-far in-domain generalization, while `ood_test` is never used for model selection.

Following the benchmark construction protocol, each synthetic task is generated from 5,000 samples. In the task files used by our implementation, 4,000 samples are exposed through the training split and the remainder are reserved for held-out test evaluation. Across domains, Biology tasks share the mapping $(t, P) \mapsto dP/dt$, whereas Material tasks share $(\epsilon, T) \mapsto \sigma$; task-level variation arises entirely from the hidden symbolic law and from the associated numeric ranges encoded in the benchmark files.

B.2 Prompt construction and executable scaffold

Our prompting interface mirrors the executable structure of the benchmark while remaining intentionally minimal. Each prompt is assembled from three pieces: the task instruction, a fixed program scaffold split into `EXACT_PREFIX` and `EXACT_SUFFIX` around the editable region, and at most one in-context inspiration drawn from the current best valid program in the same group. The implementation hard-caps the number of inspirations at one. At decoding time, only the code between `# EVOLVE-BLOCK-START` and `# EVOLVE-BLOCK-END` is extracted from the model response; the full runnable program is then reconstructed automatically from the fixed prefix, the generated block, and the fixed suffix. To keep the search space comparable across runs, the only permitted third-party libraries are `numpy`, `scipy`, and `scikit-learn`.

````
Task: <task instruction>

Improve the solution:

Score: <score>
Metrics:
  acc01: <value>
  combined_score: <value>
  ...

Code:
```python
<best prior program in the current group>
```

Generation instruction (must follow exactly):
1) Only the code between # EVOLVE-BLOCK-START
    and # EVOLVE-BLOCK-END is extracted.
2) The final program is reconstructed as
    EXACT_PREFIX + evolved_block + EXACT_SUFFIX.
3) Keep marker lines exactly as written.
4) Return one Python code block that includes
    both EVOLVE-BLOCK markers.

EXACT_PREFIX (kept unchanged):
```python
<prefix ending with # EVOLVE-BLOCK-START>
```

EXACT_SUFFIX (kept unchanged):
```python
<suffix starting from # EVOLVE-BLOCK-END>
```

Available packages: numpy, scipy, scikit-learn
````

Shared initialization. All 49 Biology and Material tasks use the same initialization scaffold. The initializer is deliberately weak: it fits an affine baseline to the raw inputs by least squares and therefore

serves as a domain-agnostic starting point shared across all runs and all groups.

```
# EVOLVE-BLOCK-START
"""Initial program: affine symbolic-regression
    baseline fit by least squares."""

import numpy as np


def scaling_law_func(data_points, params):
    X = np.atleast_2d(np.asarray(data_points,
    dtype=float))
    params = np.asarray(params, dtype=float)

    if params.ndim == 1:
        params = params[None, :]

    n_features = X.shape[1]
    expected = n_features + 1
    if params.shape[1] < expected:
        pad = np.zeros((params.shape[0],
    expected - params.shape[1]), dtype=float)
        params = np.concatenate([params, pad],
    axis=1)

    weights = params[:, :n_features]
    bias = params[:, n_features]

    pred = X @ weights.T + bias[None, :]
    return pred[:, 0] if pred.shape[1] == 1 else
     pred


def fit_scaling_law(data_points, loss_values):
    X = np.atleast_2d(np.asarray(data_points,
    dtype=float))
    y = np.asarray(loss_values, dtype=float)

    if y.ndim == 1:
        y = y[:, None]

    design = np.concatenate([X, np.ones((X.shape
    [0], 1), dtype=float)], axis=1)
    coeffs, _, _, _ = np.linalg.lstsq(design, y,
     rcond=None)

    params = coeffs.T
    return params[0] if params.shape[0] == 1
    else params
# EVOLVE-BLOCK-END
```

B.3 Grouped TTS search and implementation details

Algorithm 1 summarizes the grouped tree-structured search used throughout our experiments. Relative to the notation in the main text, the implementation uses (n, k, g) for population size, branching factor, and number of groups, respectively. The synchronized search depth is

$$T = \max\left(1, \left\lfloor \frac{N}{nk} \right\rfloor\right),$$

and the total width expanded per synchronized iteration is $w = nk$.

Although the evaluator also reports NMSE, R^2, and a benchmark-specific `combined_score`, the search objective is always training $\mathrm{Acc}_{0.1}$. In the released PIE implementation, parent selection is parameter free: scores are clipped to $[-10, 10]$ for numerical stability and sampled at unit temperature. All generations and evaluator calls are dispatched asynchronously under global concurrency caps, and every evaluation is executed in a fresh Python subprocess with a 600-second timeout.

B.4 Sweep grid and runtime settings

Unless otherwise stated, all execution uses temperature 0.7, `max_tokens=16384`, three random seeds $\{0, 1, 2\}$, and global concurrency caps of 128 LLM calls and 128 evaluator subprocesses. The main paper reports `gpt-oss-20b` on Bio and Material, and `Qwen3-30B-A3B-Thinking-2507` on Bio as a cross-backbone check. Grouping ablations are run only for `gpt-oss-20b`. Held-out `test` performance is measured after every synchronized iteration, but it is never fed back into the search loop.

B.5 Ground-truth symbolic laws for the studied domains

For completeness, we reproduce the benchmark ground-truth equations for all 49 LSR-Synth tasks considered in this paper. The task identifiers follow the naming convention in LLM-SRBench Table 4.

C Additional experimental results

C.1 Empirical budget–width trend

We sweep PBeam and PIE over total budgets $N \in \{1, 2, 4, 8, 16, 32, 64, 128\}$ and report means over three seeds after first averaging over all tasks within each domain. We fix $g = 1$ when studying the width frontier. Because this analysis contains only eight discrete budget levels, we use it to support qualitative conclusions about compute allocation rather than to claim a universal scaling law.

The width frontier Whenever we analyze $w^\star(N)$, the frontier is defined as the best attainable score at a given pair (N, w) after optimizing over all decompositions (n, k) within PBeam and within PIE and then comparing the two algorithms. Therefore the optimal width is *not* tied to a single control flow. The algorithm shown in Appendix Table 5 is simply the method that happens to attain that envelope at a given budget. When PBeam and PIE are theoretically identical, namely greedy search

Algorithm 1 Grouped TTS search used in our experiments

Require: problem p, budget N, population size n, branching factor k, groups g, algorithm $\in$ $\{\text{PBeam}, \text{PIE}\}$

$T \leftarrow \max(1, \lfloor N/(nk) \rfloor)$, $s \leftarrow n/g$
evaluate the shared initial program on the training split to obtain node x_0
initialize memory $\mathcal{M}_0$ by replicating x_0 into each of the g groups with s slots per group
for $t = 1$ to T **do**
    sample s parents per group
    PBeam: choose the top-s valid nodes by train $\mathrm{Acc}_{0.1}$
    PIE: sample without replacement from a softmax over clipped scores
    for all selected parent x **do**
        for $j = 1$ to k **do**
            build a prompt from the task instruction, the fixed scaffold, and the best valid group-local inspiration
            query the LLM and extract only the code inside the `EVOLVE-BLOCK`
            evaluate the reconstructed program on the training split; set its search score to train $\mathrm{Acc}_{0.1}$
        end for
    end for
    if algorithm $=$ PBeam **then**
        in each group, keep the top-s nodes from $\mathcal{M}_{t-1} \cup \{\text{new children}\}$
    else
        append children to memory and cap each group to $\max(4s, N)$ nodes
    end if
    evaluate the current best-so-far node on the held-out `test` split for logging only
end for
return best-scoring node seen during the run

$(n, k) = (1, 1)$ or single-step search ($T = 1$), missing PIE values are copied from PBeam. Based on this unified definition, the frontier lets us examine how the preferred width changes with budget after controlling for algorithm choice and the (n, k) decomposition. The main conclusion is qualitative: the best width tends to increase with budget, although the exact argmax can be sensitive to finite-sample noise and nearby widths often perform similarly.

Why we do not claim a universal width law. Although the optimal widths in Table 5 show a clear upward trend, the evidence is not sufficient to support a precise parametric law. First, the frontier contains only eight budget levels, all of which are powers of two and bounded by $N \leq 128$. Second, the optimal width is an argmax over noisy empirical measurements; when multiple widths are close, small seed-level fluctuations can change the selected width. Third, the trend may depend on verifier quality, model family, task difficulty, and the available parallel hardware.

For these reasons, we use the table as a practical allocation guide rather than as a scaling-law claim. A robust deployment strategy is to run a small pilot sweep over candidate widths, preferably powers of two, and then choose the smallest width whose performance is close to the observed frontier. This preserves most of the quality gains while avoiding over-interpretation of a small number of fitted points.

Performance under tuned width. When we select the best-performing width at each budget, training accuracy improves with additional compute but exhibits diminishing returns. We report this as a descriptive pattern rather than fitting a parametric power law, since the number of budget levels is small and the high-budget accuracies are already close to saturation. This result is still useful for compute allocation: most of the benefit comes from moving away from poorly matched width–depth regimes, while further gains at large budgets become progressively smaller.

Table 2: **Experimental grid used in the reported study.**

Study stage	Configurations
nk assignment	Main width–depth sweep. For each reported model/domain/seed tuple, we fix $g = 1$ and enumerate all power-of-two pairs (n, k) satisfying $nk \leq 128$. This yields 36 PBEAM configurations. PIE uses the same grid except for the degenerate $(n, k) = (1, 1)$ case and the single-step settings with $nk = 128$, yielding 27 additional configurations. Each run is launched once at maximum budget 128, and lower target budgets $N \in \{1, 2, 4, 8, 16, 32, 64, 128\}$ are recovered from best-so-far checkpoints by selecting the last logged state with cumulative `budget_used` $\leq N$.
G assignment	Grouping ablation, performed only for `gpt-oss-20b` on Bio and Material. We fix $N = 128$ and sweep group counts on the competitive high-width slices identified in the first stage. Width-32 settings are $(n, k) \in \{(32, 1), (16, 2), (8, 4)\}$ and width-16 settings are $(n, k) \in \{(16, 1), (8, 2)\}$. For each pair we test every divisor g of n in $\{1, 2, 4, 8, 16, 32\}$, producing 24 configurations per algorithm.

C.2 The study of second-order effect allocation choices

Does the decomposition into n and k matter? Once the width $w = nk$ is fixed, the remaining question is how much of that width should come from expanding more parents (n) versus generating more children per parent (k). To isolate this effect, we fix the total budget to $N = 128$, the global width to $w = 32$, and grouping to $g = 1$, and compare all admissible power-of-two decompositions of 32. Within each domain and algorithm, we group these decompositions into three coarse regimes: *population-heavy* ($n > k$), *balanced* ($n \approx k$), and *branching-heavy* ($k > n$). Table 6 reports the best (n, k) within each regime.

The main pattern is that, after choosing the right width, the exact (n, k) split is a second-order decision. Across the four domain–algorithm pairs, different regimes attain the best score, and most alternatives remain within about one percentage point of the regime optimum, with one larger outlier on Bio for PIE. This variation (typically $< 1\%$) is much smaller than the cross-width effect reported in the main text: moving to a poorly chosen width can cost several percentage points even after optimizing over (n, k) and the controller (can reach $\sim$5%). In practice, this suggests a simple recipe: first tune the width w, then use a moderate decomposition such as balanced or population-heavy as a robust default rather than over-optimizing the exact n–k split.

Does the choice between PBeam and PIE matter? We compare PIE and PBeam under the same budget N and width w with performance envelop over (n, k) combination and plot the performance gap (PIE $-$ PBeam) in Figure 4. The accuracy differences tightly fluctuate around the zero baseline, typically bounded within a marginal ± 2 percentage points. This algorithmic variance can be safely ignored when compared to the 10 to 20 point penalty of choosing the wrong width. This confirms that PIE and PBeam are largely interchangeable, reinforcing our core thesis that the width w determines the performance ceiling.

Comparison with OpenEvolve. To isolate the effect of compute allocation from that of additional engineering heuristics, we compare PBeam and PIE against OpenEvolve under strictly matched global budgets. In the reduced (n, k, g) parameterization introduced in the main text, OpenEvolve corresponds to a *fully partitioned* special case:$(n, k, g) = (w, 1, w)$, so that each of the $g = w$ groups (islands) contains exactly one active parent, generates exactly one child per synchronized iteration, and therefore operates with local population $n/g = 1$ and local width $w/g = 1$. For a fixed global budget N and width w, this gives OpenEvolve the same synchronized depth as our controllers,$T = \max\left(1, \left\lfloor \frac{N}{w} \right\rfloor\right)$, while differing only in how the width is organized within each iteration. This yields a clean matched-width comparison[3] against PBeam and PIE, which can realize the same global width with less aggressive partitioning (i.e., smaller g and/or $k > 1$). As shown in fig. 5, PBeam and PIE generally achieve better train $\mathrm{Acc}_{0.1}$ than OpenEvolve at the same (N, w) when using a mild $g = 2$ and balanced (n, k) assignment. This is consistent with our grouping analysis in section 3.2: fully partitioning the search into w isolated groups can reduce useful competition across candidates and push each group into an overly narrow local search regime. Overall, this comparison supports our main claim that, on this benchmark, the dominant gains come from getting the global compute allocation right, whereas

[3] We use the official implementation of OpenEvolve with the same prompt template in section B. Unless specially mentioned, the other hyperparameters are set as default values.

Table 3: **Biology-domain LSR-Synth equations used in this paper.** Each entry reports the right-hand side of $\frac{dP}{dt} = f(t, P)$.

ID	Right-hand side	ID	Right-hand side
BPG1	$r\left(1-\frac{P(t)}{K_0}\right)P(t) + rP(t)^{0.33}$	BPG13	$\beta P(t)\sin(\omega t) + r\left(1-\frac{P(t)}{K_0}\right)P(t)$
BPG2	$rP(t)\exp(-\gamma t) + \frac{rP(t)^2}{\alpha P(t)+1}$	BPG14	$r\left(-1+\frac{P(t)}{\alpha}\right)\left(1-\frac{P(t)}{K_0}\right)P(t) + rP(t) + \frac{rP(t)}{1+\exp(-\alpha(-\beta+P(t)))}$
BPG3	$\beta P(t)\sin(\omega t) + rP(t)\exp(-\gamma t)$	BPG15	$r\left(1-\frac{P(t)}{K_0}\right)P(t) + r\left(1-\exp(-\gamma P(t))\right)P(t) + rP(t)\exp(-\gamma t)$
BPG4	$r\left(-1+\frac{P(t)}{\alpha}\right)\left(1-\frac{P(t)}{K_0}\right)P(t) + r\left(1-\exp(-\gamma P(t))\right)P(t)$	BPG16	$rP(t)^{0.33} + rP(t)\exp(-\gamma t)$
BPG5	$r\left(1-\frac{P(t)}{K_0}\right)P(t) + \frac{rP(t)}{1+\exp(-\alpha(-\beta+P(t)))}$	BPG17	$r\left(-1+\frac{P(t)}{\alpha}\right)\left(1-\frac{P(t)}{K_0}\right)P(t) + rP(t)^{0.33} + rP(t)$
BPG6	$r\left(1-\frac{P(t)}{K_0}\right)P(t) + \frac{rP(t)^2}{\alpha P(t)+1}$	BPG18	$r\left(-1+\frac{P(t)}{\alpha}\right)\left(1-\frac{P(t)}{K_0}\right)P(t) + rP(t)^{0.33}$
BPG7	$-Q\alpha P(t) + r\left(1-\frac{P(t)}{K_0}\right)P(t) + rP(t)^{0.33} + rP(t)$	BPG19	$\beta P(t)\sin(\omega t) + r\left(1-\frac{P(t)}{K_0}\right)P(t) + rP(t)$
BPG8	$r\left(-1+\frac{P(t)}{\alpha}\right)\left(1-\frac{P(t)}{K_0}\right)P(t) + r\left(1-\frac{P(t)}{K_0}\right)P(t) + rP(t)^{0.33}$	BPG20	$r\left(1-\frac{P(t)}{K_0}\right)P(t) + \frac{rP(t)}{t^\alpha}$
BPG9	$r\left(1-\frac{P(t)}{K_0}\right)P(t) + rP(t)^{0.33} + rP(t)$	BPG21	$r\left(-1+\frac{P(t)}{\alpha}\right)\left(1-\frac{P(t)}{K_0}\right)P(t) + r\left(1-\frac{P(t)}{K_0}\right)P(t) + \frac{rP(t)}{1+\exp(-\alpha(-\beta+P(t)))}$
BPG10	$r\left(-1+\frac{P(t)}{\alpha}\right)\left(1-\frac{P(t)}{K_0}\right)P(t) + r\left(1-\frac{P(t)}{K_0}\right)P(t) + r\left(1-\exp(-\gamma P(t))\right)P(t)$	BPG22	$r\left(-1+\frac{P(t)}{\alpha}\right)\left(1-\frac{P(t)}{K_0}\right)P(t) + \frac{rP(t)}{t^\alpha}$
BPG11	$rP(t)^{0.33} + rP(t)$	BPG23	$r\left(1-\exp(-\gamma P(t))\right)P(t) + rP(t)\exp(-\gamma t)$
BPG12	$r\left(1-\frac{P(t)}{K_0}\right)P(t) + rP(t)^{0.33} + rP(t)\exp(-\gamma t)$	BPG24	$r\left(1-\frac{P(t)}{K_0}\right)P(t) + r\left(1-\exp(-\gamma P(t))\right)P(t)$

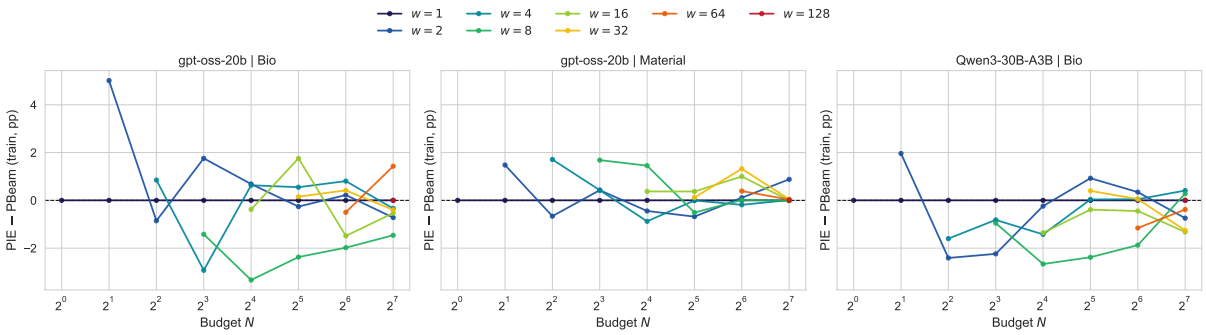

Figure 4: **PIE − PBeam after optimizing (n, k) inside each algorithm.** The vertical axis shows train $\mathrm{Acc}_{0.1}$ differences in percentage points. The effect size is much smaller than the effect of choosing the right width.

Table 4: **Material-domain LSR-Synth equations used in this paper.** Each entry reports the corresponding stress law $\sigma(\epsilon, T) = g(\epsilon, T)$.

ID	Right-hand side	ID	Right-hand side
MatSci1	$E_0\epsilon\left(-\alpha_T(T-T_0)+1\right)-\beta(T-T_0)+\epsilon^M\eta(T-T_0)$	MatSci14	$-\beta(T-T_0)+\epsilon\eta\exp\left(-(T-T_0)^2\right)$
MatSci2	$H\epsilon^3+K\epsilon^N\exp\left(-\frac{Q}{RT}\right)+\epsilon\eta\sin(T-T_0)$	MatSci15	$-\beta(T-T_0)+\epsilon^M\eta(T-T_0)$
MatSci3	$H\epsilon^3+\eta(T-T_0)\exp(-\epsilon)$	MatSci16	$E_0\epsilon\left(-\alpha_T(T-T_0)+1\right)+\epsilon\eta\exp\left(-(T-T_0)^2\right)$
MatSci4	$H\epsilon^3+K\epsilon^N\exp\left(-\frac{Q}{RT}\right)+\epsilon^3\eta(T-T_0)$	MatSci17	$E_0\epsilon^2+\epsilon\eta(T-T_0)^2$
MatSci5	$E_0\epsilon^2+\eta(T-T_0)\log(\epsilon+1)$	MatSci18	$E_0\epsilon\left(-\alpha_T(T-T_0)+1\right)-\beta(T-T_0)+\eta(T-T_0)\log(\epsilon+1)$
MatSci6	$E_0\epsilon\left(-\alpha_T(T-T_0)+1\right)+K\epsilon^N\exp\left(-\frac{Q}{RT}\right)+\epsilon^M\eta(T-T_0)$	MatSci19	$H\epsilon^3+\eta(T-T_0)\sin(\epsilon)$
MatSci7	$E_0\epsilon\left(-\alpha_T(T-T_0)+1\right)+\epsilon\eta(T-T_0)^2$	MatSci20	$E_0\epsilon^2-\beta(T-T_0)+\epsilon^3\eta(T-T_0)$
MatSci8	$H\epsilon^3-\beta(T-T_0)+\eta(T-T_0)\log(\epsilon+1)$	MatSci21	$E_0\epsilon^2+\epsilon\eta\sin(T-T_0)$
MatSci9	$E_0\epsilon\left(-\alpha_T(T-T_0)+1\right)+\epsilon^M\eta(T-T_0)$	MatSci22	$K\epsilon^N\exp\left(-\frac{Q}{RT}\right)-\beta(T-T_0)+\eta(T-T_0)\log(\epsilon+1)$
MatSci10	$H\epsilon^3-\beta(T-T_0)+\epsilon^3\eta(T-T_0)$	MatSci23	$E_0\epsilon\left(-\alpha_T(T-T_0)+1\right)+H\epsilon^3+\eta(T-T_0)\sin(\epsilon)$
MatSci11	$H\epsilon^3+K\epsilon^N\exp\left(-\frac{Q}{RT}\right)+\epsilon\eta(T-T_0)^2$	MatSci24	$K\epsilon^N\exp\left(-\frac{Q}{RT}\right)+\epsilon\eta\sin(T-T_0)$
MatSci12	$K\epsilon^N\exp\left(-\frac{Q}{RT}\right)+\epsilon^3\eta(T-T_0)$	MatSci25	$E_0\epsilon^2+E_0\epsilon\left(-\alpha_T(T-T_0)+1\right)+\eta(T-T_0)\log(\epsilon+1)$
MatSci13	$E_0\epsilon\left(-\alpha_T(T-T_0)+1\right)+K\epsilon^N\exp\left(-\frac{Q}{RT}\right)+\epsilon\eta\exp\left(-(T-T_0)^2\right)$		

Table 5: **Exact optimal width on `gpt-oss-20b` without grouping.** For each pair (N, w), we first optimize over all decompositions (n, k) inside PBeam and inside PIE, and then choose the better algorithm. The table therefore reports which algorithm attains the envelope at each budget rather than a globally preferred control flow.

	Bio			Material		
Budget N	Algorithm	(n, k, T)	$w^\star$	Algorithm	(n, k, T)	$w^\star$
1	PBeam	$(1, 1, 1)$	1	PBeam	$(1, 1, 1)$	1
2	PBeam	$(1, 1, 2)$	1	PBeam	$(1, 1, 2)$	1
4	PBeam	$(1, 2, 2)$	2	PBeam	$(2, 1, 2)$	2
8	PIE	$(1, 2, 4)$	2	PIE	$(2, 2, 2)$	4
16	PIE	$(1, 2, 8)$	2	PBeam	$(1, 4, 4)$	4
32	PBeam	$(1, 8, 4)$	8	PBeam	$(1, 4, 8)$	4
64	PBeam	$(2, 8, 4)$	16	PIE	$(8, 2, 4)$	16
128	PBeam	$(4, 8, 4)$	32	PIE	$(64, 1, 2)$	64

Table 6: **Best (n, k) decomposition within each regime at fixed budget $N = 128$, width $w = 32$, and no grouping ($g = 1$).** For each domain and algorithm, we partition the six admissible decompositions of $w = 32$ into three regimes and report the best (n, k) and its train $\mathrm{Acc}_{0.1}$ within each regime. Relative gap is computed against the best regime under the same domain and algorithm.

Domain	Algorithm	Regime	Best (n, k)	Train $\mathrm{Acc}_{0.1}$	Rel. gap
Bio	PBeam	Population-heavy	$(16, 2)$	0.9816	0.28%
Bio	PBeam	**Balanced**	$(4, 8)$	**0.9844**	**0.00%**
Bio	PBeam	Branching-heavy	$(1, 32)$	0.9801	0.44%
Bio	PIE	**Population-heavy**	$(16, 2)$	**0.9804**	**0.00%**
Bio	PIE	Balanced	$(8, 4)$	0.9612	1.95%
Bio	PIE	Branching-heavy	$(2, 16)$	0.9794	0.10%
Material	PBeam	Population-heavy	$(16, 2)$	0.9857	1.15%
Material	PBeam	**Balanced**	$(8, 4)$	**0.9971**	**0.00%**
Material	PBeam	Branching-heavy	$(1, 32)$	0.9876	0.95%
Material	PIE	Population-heavy	$(32, 1)$	0.9884	0.92%
Material	PIE	Balanced	$(4, 8)$	0.9975	$< 0.01\%$
Material	PIE	**Branching-heavy**	$(2, 16)$	**0.9976**	**0.00%**

Regime definitions: population-heavy $\{(32, 1), (16, 2)\}$, balanced $\{(8, 4), (4, 8)\}$, and branching-heavy $\{(2, 16), (1, 32)\}$.

additional heuristics such as island engineering, crossover, or specialized prompting appear to be secondary.

C.3 Wall-clock time speedups through grouping

As claimed in the main paper, grouping improves inference efficiency via reduced synchronization overhead. Intuitively, one might hypothesize that evaluating candidates in larger groups could artificially enforce search diversity—by allowing parallel branches to explore independently before being evaluated and pruned—which might in turn elevate the overall reasoning performance. However, our empirical results reveal that this is generally not the case: grouping predominantly acts as a strict accuracy-efficiency trade-off rather than an algorithmic performance booster.

To isolate the exact impact of grouping once the search width is already chosen, Figure 6 fixes the compute budget at $N = 128$ and compares the optimal grouped configurations ($g > 1$) against their exact ungrouped baselines ($g = 1$) of the same width.

On the Material domain, grouping can occasionally be helpful: at width 16, moving from $g = 1$ to $g = 2$ increases the train $\mathrm{Acc}_{0.1}$ by 0.295 percentage points and yields a $1.043\times$ speedup, while $g = 8$ still preserves a slight accuracy gain alongside a $1.075\times$ speedup. In contrast, the Bio domain is more fragile. At width 16, setting $g = 8$ provides a $1.053\times$ speedup but suffers a severe accuracy penalty, losing 1.54 percentage points. Ultimately, while grouping delivers measurable system-level acceleration, overly large g should be used cautiously.

C.4 Generalization: internal train–test gap and cross-model consistency

Generalization from training to test sets As discussed in the main paper, we determine the search configuration (e.g., the search width $w^\star$) using training-set performance. As shown in Figure 7, the test accuracy closely tracks the training accuracy across all compute budgets, indicating no obvious overfitting. However, optimizing strictly on the training set at larger budgets can occasionally lead to a slight performance drop on the unseen test set (e.g., $N \geq 2^5$ in the Material domain). Furthermore, our control flow is designed to greedily maximize the training accuracy at each iteration and strictly retain the highest-scoring paths during each step of the beam search. While this aggressive, step-wise optimization guarantees peak performance on the training distribution, it inherently risks mild overfitting. Consequently, the resulting configurations may yield slightly sub-optimal accuracy on the unseen test set.

Does the optimal width law generalize across backbones? To investigate whether our empirically derived width scaling behavior is tied to a specific model architecture, we evaluate the `Qwen3-30B-A3B` backbone on the Bio domain. As illustrated in Figure 8, the qualitative width–budget trend is similar across the two model families. The exact optimal width trajectories for both `Qwen3` and

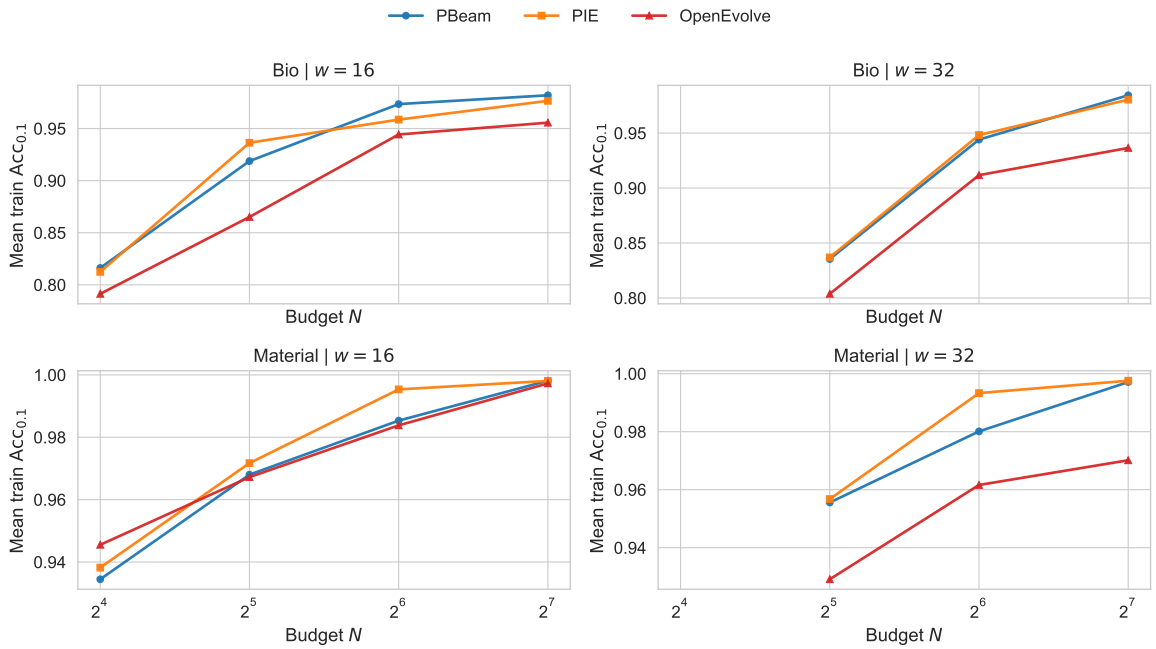

Figure 5: Matched comparison among PBeam, PIE, and OpenEvolve.

`gpt-oss` perfectly overlap at early compute budgets ($N \leq 8$). The only localized divergence occurs at the intermediate budget of $N = 16$, where Qwen temporarily favors a deeper search ($w^\star = 2$) while the GPT backbone expands to a wider allocation ($w^\star = 8$). This aligned trajectory suggests that the width–budget trade-off is not unique to one backbone, although the exact preferred width remains model- and budget-dependent.

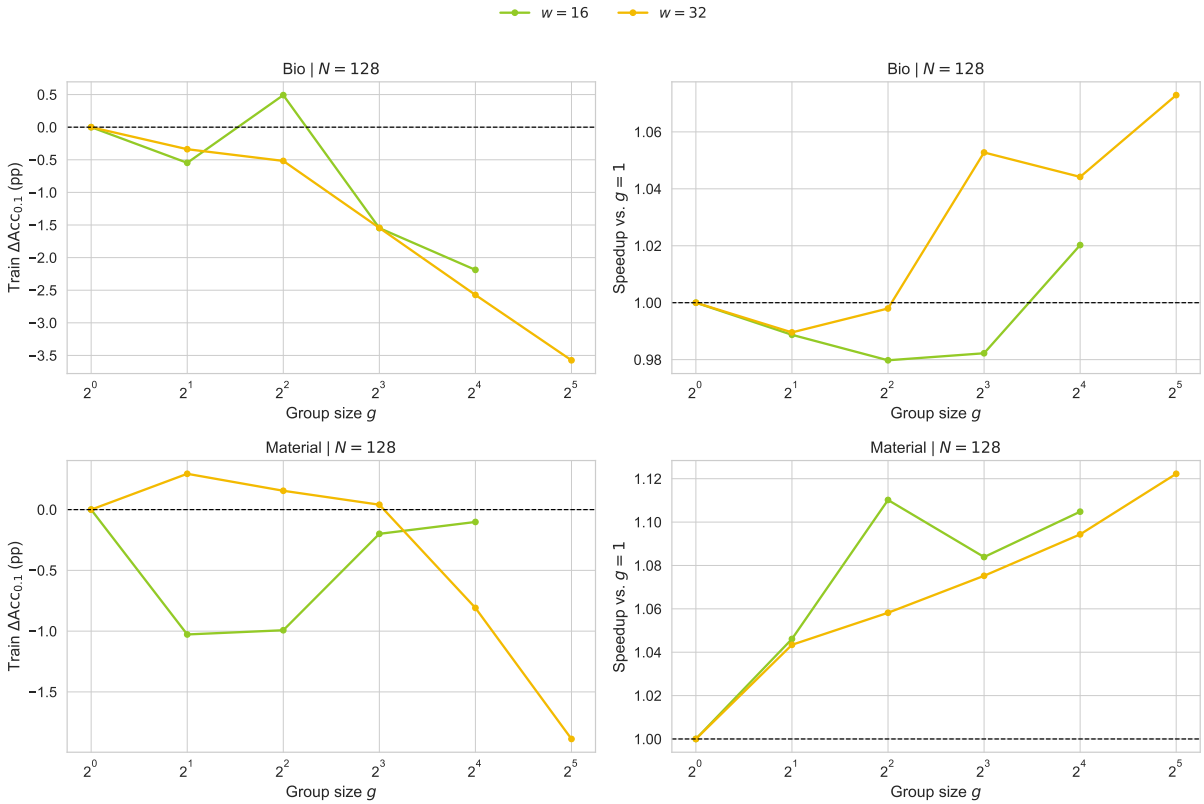

Figure 6: **Effect of group size on performance and wall-clock time.** For each domain and width, every point is the best measured configuration at fixed (w, g) and $N = 128$, compared against the best ungrouped configuration with the same width. Moderate grouping can improve throughput with modest quality loss, but excessively large g eventually collapses the per-group width and hurts performance.

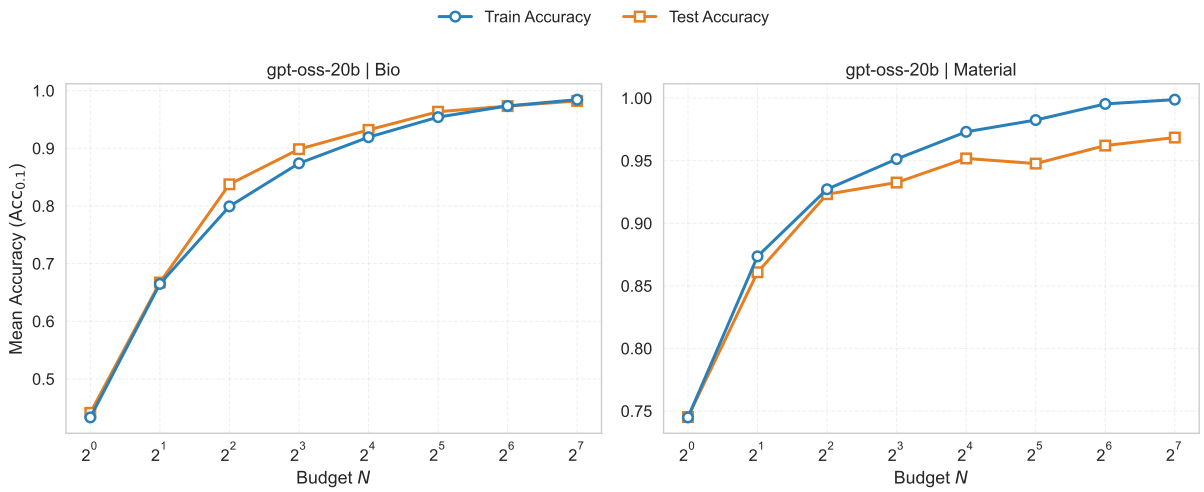

Figure 7: **Train vs. test performance of gpt-oss-20b.** Mean accuracy ($\mathrm{Acc}_{0.1}$) on the training and test sets for the Bio (left) and Material (right) domains across varying compute budgets N.

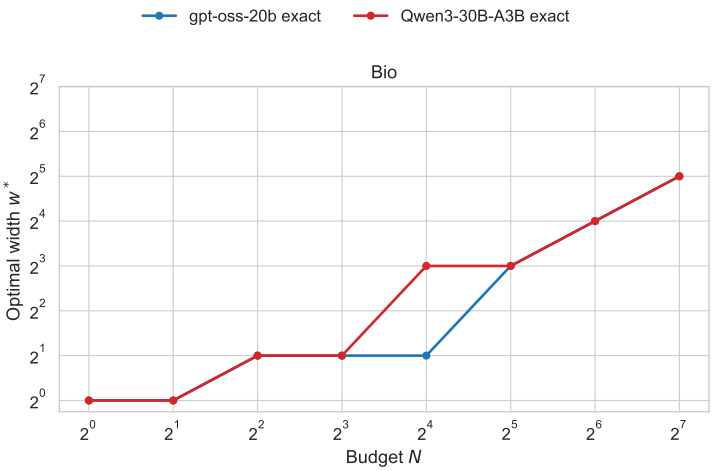

Figure 8: **Auxiliary cross-model comparison on Bio with `Qwen3-30B-A3B`.** Solid lines show the exact best-performing width in the sweep. The same qualitative width–depth trade-off appears across models, although the exact mid-budget operating point remains backbone-dependent.