Agentic Reverse Engineering: From I/O to Code with Discovery Agent

Yihong Dong[1,2], **Jiaru Qian**[3], **Haoran Zhang**[4], **Peixu Wang**[5], **Binhua Li**[2], **Zhi Jin**[3]
Yongbin Li[2], **Ge Li**[3], **Xiaokang Yang**[1], **Xue Jiang**[2,3]
[1] Shanghai Jiao Tong University [2] Tongyi Lab, Alibaba Group
[3] Peking University [4] Renmin University of China [5] National University of Singapore

Abstract

The automatic synthesis of a program from any form of specification is regarded as a holy grail of computer science. Fueled by LLMs, NL2Code has achieved tremendous success, yet the fundamentally more challenging task of synthesizing programs from input-output behavior, which we refer to as IO2Code, remains largely unsolved. Whereas NL2Code can exploit the semantic alignment between natural language and code acquired during pretraining, IO2Code requires recovering underlying principles from concrete computational behavior, navigating a vast and underspecified hypothesis space. To address this, we propose DIO-Agent, a discovery agent for IO2Code. Our method frames IO2Code as an evolutionary search over discrete program space, in which an LLM serves as the mutation operator and concrete error signals from execution guide each mutation. To prevent the search from wandering into structurally complex yet incorrect dead ends, we introduce the Transformation Priority Premise as a mutation prior that biases the LLM toward the simplest hypothesis consistent with current evidence, progressively escalating from constants to conditionals to iteration only when simpler constructs are insufficient. To facilitate systematic study, we further construct an IO2CodeBench spanning multiple difficulty levels. Extensive experiments show that DIO-Agent consistently outperforms both traditional program-by-example method and SOTA evolution-agent baselines across all difficulty levels and various LLMs, while substantially surpassing test-time scaling strategies with equivalent sampling budgets.[1]

1 Introduction

Large language models (LLMs) have achieved remarkable success in code generation [Chen et al., 2021, Nijkamp et al., 2022, Jiang et al., 2024, Dong et al., 2024], largely built upon the NL2Code formulation [Iyer et al., 2018, Lu et al., 2021, Austin et al., 2021] that converts natural-language descriptions into executable programs [Hendrycks et al., 2021]. NL2Code relies on human-provided descriptions, limiting its applicability when such descriptions are unavailable or unreliable. However, a wide range of real-world tasks requires developers and researchers to work not from NL descriptions, but from computational behavior. From migrating legacy systems with lost source code and reverse-engineering black-box APIs, to abducing algorithms from scientific observations and capturing user intent through input-output demonstrations, these scenarios span system evolution, scientific discovery, and human-computer interaction [Collie et al., 2020]. We refer to this class of tasks as IO2Code, where the goal is to synthesize programs from input-output behavior(Figure 1).

IO2Code poses fundamentally more severe challenges than NL2Code. NL2Code largely reduces to composing and reusing code patterns aligned with natural-language descriptions learned during

[1] Our code and dataset are available at `https://github.com/JiaruQian/IO2Code`.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

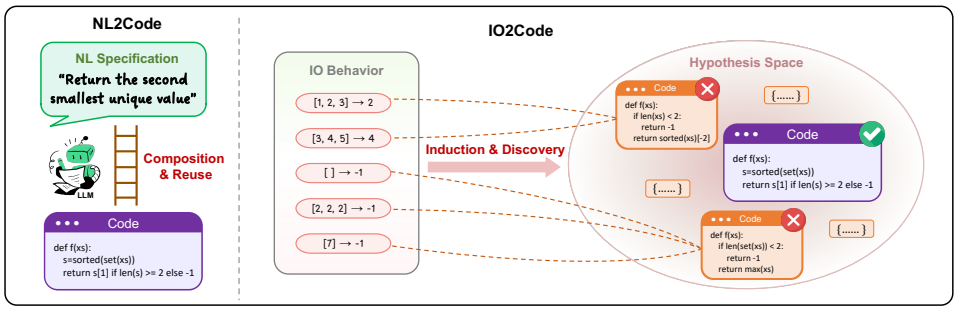


Figure 1: Illustration of IO2Code, where the mainstream NL2Code task evaluates the composition and reuse ability of LLMs, while the IO2Code task evaluates the induction and discovery ability.

pretraining [Zan et al., 2023, Feng et al., 2020, Wang et al., 2021]. In contrast, IO2Code demands induction and discovery, where the model must uncover latent algorithmic structures from concrete computational behavior, infer causal relationships between inputs and outputs, and autonomously construct complete program logic. This requires abductive reasoning from the specific to the general, in which the composition and reuse strategies effective in NL2Code become inapplicable. Furthermore, input-output observations are inherently underspecified, as multiple programs can be consistent with a finite set of examples, demanding that the agent navigate a vast hypothesis space while avoiding overfitting to spurious patterns.

To tackle the challenge of IO2Code, we draw on a well-established observation from software engineering. In the practice of Test-Driven Development (TDD), the Transformation Priority Premise (TPP) [Martin, 2021] shows that programmers construct algorithms that pass all tests most effectively when they apply transformations in order of increasing complexity, progressing from constants to variables, from unconditional statements to conditionals, and from conditionals to loops or recursion, and that violating this ordering frequently leads to impasses requiring costly rewrites. We observe that TPP can be repurposed as a structural constraint on autonomous program discovery. In IO2Code, where no specification exists beyond input-output behaviors, such a complexity ordering offers a principled mechanism to force the search process to exhaust simpler hypotheses before entertaining structurally more complex ones, thereby preventing both overfitting to surface patterns and aimless wandering in the program space.

In this paper, we propose DIO-Agent, a discovery agent for IO2Code. Our method formulates the task as an evolutionary search in which an LLM acts as the mutation operator. In the discrete program space without natural-language guidance, allowing a model to mutate code blindly can easily lead it into structurally complex yet incorrect dead ends. To address this, we introduce TPP as a mutation prior that biases the search toward the simplest hypothesis first, escalating to more expressive constructs only when simpler ones fail to explain the observed behavior. This evolutionary process is organized as a curriculum of increasing difficulty, with concrete error signals from execution grounding each mutation in debugging evidence rather than scalar rewards. To facilitate research, we establish an IO2CodeBench spanning multiple difficulty levels from basic data operations to complex algorithmic reasoning.

We conduct extensive experiments on the proposed IO2CodeBench across all difficulty levels. The results demonstrate that DIO-Agent achieves consistent improvements over existing approaches, including traditional PBE methods and SOTA evolution-agent baselines, while substantially surpassing test-time scaling strategies with equivalent sampling budgets. Moreover, DIO-Agent yields stable performance gains across diverse base models, confirming its generalizability. We further validate the effectiveness of each component through comprehensive ablation studies, hyperparameter sensitivity analyses, and illustrative case studies.

2 Motivation Example

To illustrate the challenges of IO2Code and motivate our approach, consider a concrete example. Suppose an agent is presented with the following input-output pairs and asked to synthesize a program `f` consistent with all of them:

```
f(1) = []       f(2) = [2]         f(3) = [3]          f(4) = [2, 2]
f(6) = [2, 3]   f(8) = [2, 2, 2]   f(12) = [2, 2, 3]   f(30) = [2, 3, 5]
```

To human with mathematical training, the underlying rule is recognizable: `f` computes prime factorization of its input. But the agent receives no such hint. It has no access to the function name, docstring, or NL description, and must recover the rule purely from computational behavior.

The Overfitting Trap. The most straightforward strategy, and the one that unconstrained models tend to adopt, is to memorize the observations directly:

```
def f(n):
    if n == 1: return []
    elif n == 2: return [2]
    elif n == 3: return [3]
    ...
```

This program mirrors the tests rather than generalizing from them. It is a lookup table, not an algorithm. It will fail on any input not seen during synthesis such as `f(9)` or `f(100)`. In NL2Code, such overfitting is unlikely because the NL specification (e.g., "compute the prime factors of `n`") already constrains the model toward a general solution. In IO2Code, no such constraint exists, and the agent faces an underspecified hypothesis space in which memorization is always locally optimal.

The Blind Mutation Trap. An alternative strategy is to apply evolutionary search, randomly mutating candidate programs and selecting those that pass more test cases. However, without structural guidance, this search quickly encounters a different failure mode. The agent may produce programs that are syntactically complex yet semantically wrong. For example, a program that computes divisors rather than prime factors, or one that produces outputs of the correct length but with incorrect values. These candidates occupy complex regions of the program space that are difficult to escape through further mutation, creating structural dead ends. The combinatorial explosion of the unconstrained discrete program space makes blind search intractable for all but the simplest tasks.

From Specific to General. Both failure modes share a root cause: the absence of a strategy for navigating the hypothesis space from simple to complex. TPP [Martin, 2021] provides exactly such a strategy. In the context of test-driven development, correct algorithms emerge most reliably when programmers apply behavior-changing transformations in order of increasing structural complexity. TPP identifies a priority ordering that progresses from returning constants to variables, to conditionals, to loops, and ultimately to recursion, and demonstrates that adhering to this ordering avoids the impasses that arise when complex transformations are attempted prematurely. To see how such a principle might apply to IO2Code, consider what happens when the agent is required to follow this complexity ordering on our running example, progressing through levels of increasing structural complexity and exhausting each level before advancing to the next:

Levels 1–2: Constant and Variable. The agent begins with the simplest possible programs:

```
def f(n):
    return []
```

```
def f(n):
    return [n]
```

The constant `return []` handles only `f(1)`. Generalizing to `return [n]` covers `f(2)` and `f(3)`, but regresses on `f(1) = []` and fails on `f(4) = [2, 2]`. No variable-level program can satisfy all observations. This signals that the rule involves decomposing `n`, not merely returning it.

Level 3: Conditional. The agent introduces a conditional to split the execution path:

```
def f(n):
    factors = []
    if n % 2 == 0:
        factors.append(2)
        n = n // 2
    if n > 1:
        factors.append(n)
    return factors
```

This captures the idea of divisibility testing but only applies it once. The case `f(8) = [2, 2, 2]` reveals that a single conditional is insufficient: the operation must be repeated.

Level 4: Iteration. The agent transforms the inner `if` into a `while`:

```
def f(n):
    factors = []
    d = 2
    while d <= n:
        while n % d == 0:
            factors.append(d)
            n = n // d
        d += 1
    return factors
```

The transformation from `if` to `while` allows the resulting program to generalize to all inputs, including those never seen during synthesis.

The strategy provides three benefits for IO2Code. It prevents overfitting by forcing the agent to exhaust simpler hypotheses before resorting to case-specific logic, structures the search space into a sequence of nested and increasingly expressive subspaces, and produces informative failure patterns at each stage that reveal what additional structural complexity is needed, guiding the agent toward the next level of generalization.

3 Methodology

In this section, we first formalize the **IO2Code** task, and then describe the overview of **DIO-Agent**, finally introduce its key elements in detail, including **curriculum-wise evolution**, **transformation-priority guided mutation**, and **error-grounded feedback**.

Formalization of IO2Code. The traditional code generation task typically relies on natural language intents. In contrast, the IO2Code problem aims to synthesize a generalized program strictly from a limited set of Input-Output (I/O) pairs. Formally, an IO2Code task is defined by a set of visible examples $\mathcal{D}_{\text{vis}} = \{(x_i, y_i)\}_{i=1}^{n}$, where each x_i is an input (scalar, list, tuple, or nested structure) and y_i is the corresponding output produced by an unknown target program p^*. The objective is to synthesize a program $\hat{p}$ such that $\hat{p}(x_i) = y_i$ for all visible examples and, critically, $\hat{p}$ generalizes to held-out test cases $\mathcal{D}_{\text{test}} = \{(x_j, y_j)\}_{j=1}^{m}$ that the agent never observes during search. Fitness on the visible set is defined as:

$$f(\hat{p}) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}[\hat{p}(x_i) = y_i], \tag{1}$$

where $\mathbf{1}[\cdot]$ is the indicator function. A candidate is considered a *solution* if and only if $f(\hat{p}) = 1.0$ on the visible set and it additionally passes all held-out tests—the latter serving as a generalization check against overfitting to spurious patterns in the visible examples.

DIO-Agent Backbone. DIO-Agent addresses the IO2Code challenge with an LLM-driven evolutionary loop in which LLMs act as mutation operators and executable evaluators provide the selection signal. Search starts from an initial program template copied to multiple islands so that different search trajectories can unfold in parallel.

Within each island, DIO-Agent repeatedly performs four steps: (1) sampling a parent via a mixture of random exploration, fitness-oriented exploitation, and weighted sampling; (2) building a compact prompt context containing the parent, the two best-performing programs from the same island, and one randomly sampled inspiration program; (3) asking the LLM to propose a `SEARCH/REPLACE` diff rather than a full rewrite, preserving already working code; and (4) executing the child in a sandbox, scoring it, and inserting it back if novel and competitive. Periodic migration exchanges useful programs across islands to avoid premature convergence.

This loop is specialized through three tightly coupled design choices: Curriculum-wise Evolution organizes the process stage by stage; Transformation Priority Premise shapes how mutations are proposed within each stage; and Error-Grounded Feedback turns execution results into structured

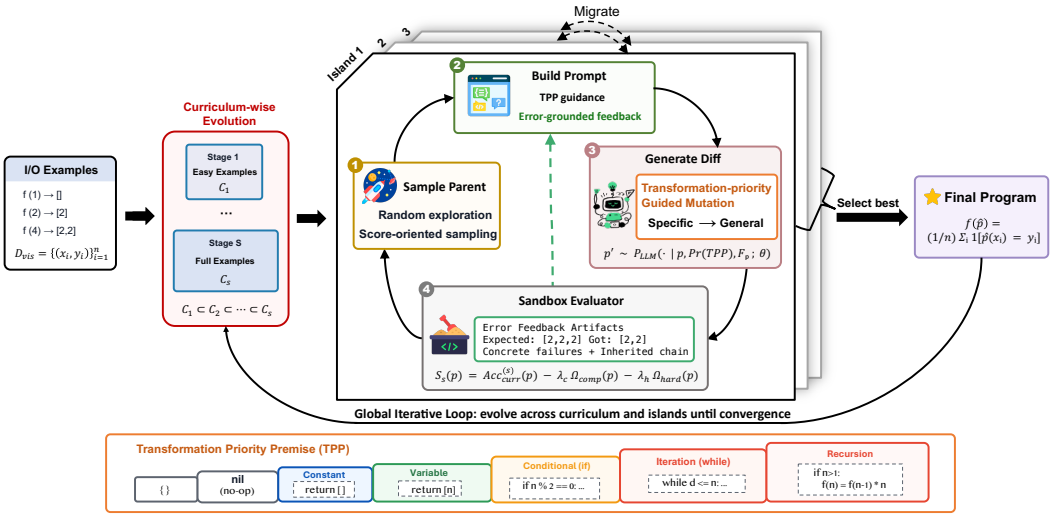


Figure 2: Overview of DIO-Agent.

feedback and adopts a fine-grained score to guide evolution. The best program of each stage is passed on as the initial program for the next. The overview of DIO-Agent is illustrated in Figure 2.

3.1 Curriculum-wise Evolution

DIO-Agent organizes the evolving process stage by stage. We first sort visible examples from easy to hard using the heuristic that favors shorter and structurally simpler inputs and outputs. This yields a nested curriculum:

$$\mathcal{C}_1 \subset \mathcal{C}_2 \subset \cdots \subset \mathcal{C}_S = \mathcal{D}_{\text{vis}}, \tag{2}$$

where each stage reveals only a prefix of the full visible set. Let $\Delta_s = \mathcal{C}_s \setminus \mathcal{C}_{s-1}$ denote the newly introduced examples at stage s.

Stage 1 starts from the initial program template of the task. For every later stage $s > 1$, DIO-Agent initializes all islands with the best program p^*_{s-1} found in stage $s-1$, rather than restarting search from scratch. Inside stage s, the mutation prompt only exposes examples in $\mathcal{C}_s$ and explicitly highlights Δ_s, while the evaluator additionally samples $\mathcal{R}_s \subseteq \mathcal{C}_{s-1}$ as replay cases.

This stage-wise decomposition changes both *what* the model must solve and *where* evolution begins. Early stages encourage simple, high-bias rules that solve easy examples; later stages broaden the behavioral specification and refine an already partially correct program. The final stage exposes all visible examples, and the resulting best program p^*_S is then evaluated once on $\mathcal{D}_{\text{test}}$.

3.2 Transformation-priority Guided Mutation

Inspired by Test-Driven Development (TDD), we introduce the Transformation Priority Premise (TPP). In a vast, non-convex code space, uniform stochastic prompting invariably leads to semantic degradation. Furthermore, evaluating a program solely by its correctness is insufficient for induction and discovery. DIO-Agent injects TPP into every mutation.

Concretely, the evolution presents an ordered preference over progressively more expressive transformations as follows: `nil`→`constant`, `constant`→`scalar`, `statement`→`statements`, `unconditional`→`if`, `scalar`→`array`, `if`→`while`, and `expression`→`function`. The LLM is also explicitly instructed to avoid exact input matching, memorized lookup tables, and other hard-coded solutions tied to seen examples. This makes the mutation operator prefer the smallest hypothesis that explains the current evidence before moving to more complex control flow.

We view this design as an evolutionary realization of the classical Red-Green-Refactor lifecycle. When the parent still fails many visible examples, the TPP prior encourages minimal edits that establish a basic working hypothesis. As more examples become satisfied, the LLM is allowed to

Table 1: Main results between DIO-Agent and seven baselines on the IO2CodeBench, where the best results are shown in **bold**.

	IO2CodeBench					
Method	Base	Algorithm	Geometry	Multimodal	Average	Token/iter
Direct	35.48	48.57	33.33	11.33	32.18	-
PBE [Gulwani, 2016]	20.97	20.00	0	17.33	14.58	-
E-PBE [Verbruggen et al., 2025]	51.61	60.00	22.22	20.67	38.63	1915.14
TDD-guided Generation	20.97	48.57	61.11	19.33	37.50	**544.90**
FunSearch [Romera-Paredes et al., 2024]	46.77	65.71	38.89	28.67	45.01	2911.99
AlphaEvolve [Novikov et al., 2025]	51.61	60.00	55.56	22.00	47.29	4130.42
CodeEvolve [Assumpção et al., 2025]	61.29	60.00	44.44	32.67	49.60	5829.32
DIO-Agent	**61.29**	**71.43**	**61.11**	**40.67**	**58.63**	3738.31

introduce branches, collections, or loops only when simpler transformations are insufficient. Once the current visible slice is already solved, the search pressure shifts toward removing unnecessary structure rather than adding more case-specific logic. In this way, TDD-style minimal generalization is not run outside evolution; it becomes the mutation prior inside the DIO-Agent loop itself.

3.3 Error-Grounded Feedback

Since many candidate programs can obtain similar accuracy scores while failing for very different reasons, DIO-Agent turns execution results into structured feedback artifacts that are passed back into subsequent mutation prompts.

For a candidate program p at curriculum stage s, the evaluator executes p on two sets: the currently visible curriculum set $\mathcal{C}_s$, and a replay set $\mathcal{R}_s$ sampled from examples introduced in earlier stages. The evaluator returns not only accuracies, but also a small number of concrete failure cases from both sets, including mismatched outputs or runtime errors. These artifacts are attached to the evaluated program and, in later iterations, are injected into the prompt of its descendants together with an inherited feedback chain from ancestor programs. As a result, the LLM receives explicit debugging evidence rather than only learning from a scalar reward.

The stage-level selection score is defined as:

$$\mathcal{S}_s(p) = \mathrm{Acc}^{(s)}_{\mathrm{curr}}(p) - \lambda_c\,\Omega_{\mathrm{comp}}(p) - \lambda_h\,\Omega_{\mathrm{hard}}(p), \tag{3}$$

where $\mathrm{Acc}^{(s)}_{\mathrm{curr}}$ is the accuracy on the current curriculum set, Ω_{comp} penalizes complex programs (e.g., Token Length or Time complexity), and Ω_{hard} is a penalty of memorization (e.g., overlap between the provided I/Os and the generated code). We set $\lambda_c = \lambda_h = 0.1$ by default.

4 Experiments

We conduct extensive experiments on IO2CodeBench spanning difficulty levels, comparing DIO-Agent against both traditional Programming-by-Example (PBE) methods and mainstream evolution-agent baselines. We also provide the generalizability across different base models, test-time scaling strategy comparison, ablation studies, and further analysis. More comprehensive experiments and detailed setups are provided in Appendix A-I.

4.1 Main Results

The main results on the IO2CodeBench are presented in Table 1. DIO-Agent achieves state-of-the-art performance across all difficulty levels, consistently outperforming both traditional PBE methods and modern evolution-agent baselines. Compared to the strongest evolutionary baseline, CodeEvolve, DIO-Agent shows superior generalization, particularly in algorithmic reasoning and geometric computation. In the Base level, DIO-Agent matches the best-performing baseline at 61.29%, demonstrating its robustness in synthesizing standard patterns. The performance gap widens significantly in harder levels. In the Algorithm level, DIO-Agent reaches 71.43%, surpassing all baselines.

Table 2: Performance Across Different LLMs.

Base LLM	Method	IO2CodeBench					
		Base	Algorithm	Geometry	Multimodal	Average	Token/iter
deepseek v3.2	Direct	35.48	48.57	33.33	11.33	32.18	-
	AlphaEvolve	51.61	60.00	55.56	22.00	47.29	4130.42
	DIO-Agent	**61.29**	**71.43**	**61.11**	**40.67**	**58.63**	**3738.31**
qwen-3.6-plus	Direct	43.55	40.00	27.77	11.33	30.66	-
	AlphaEvolve	64.52	57.14	44.44	21.33	46.86	9274.40
	DIO-Agent	**66.13**	**71.43**	**55.56**	**28.67**	**55.45**	**7068.28**
claude-sonnet-4.6	Direct	37.10	57.14	38.89	12.67	44.38	-
	AlphaEvolve	54.84	68.57	44.44	20.67	47.13	**2502.24**
	DIO-Agent	**61.29**	**65.71**	**72.22**	**30.00**	**57.31**	2785.67

In the Geometry level, DIO-Agent achieves 61.11%, a substantial improvement over CodeEvolve's 44.44%. Notably, in the Multimodal level, DIO-Agent attains 40.67%, nearly doubling AlphaEvolve (22.00%), suggesting strong capability in managing the noise inherent in multimodal information extraction. Despite its superior performance, DIO-Agent maintains competitive token efficiency. Among high-performing evolutionary methods, DIO-Agent (3738.31 tokens/iter) is considerably more efficient than CodeEvolve (5829.32) and AlphaEvolve (4130.42), indicating that our method relies on a targeted and structured search process rather than brute-force generation.

4.2 Performance Across Different LLMs

To evaluate the generalizability of our method, we conduct experiments across three representative LLMs: DeepSeek V3.2, Qwen-3.6-Plus, and Claude-Sonnet-4.6. The results in Table 2 demonstrate that DIO-Agent consistently enhances program discovery capabilities across all tested models. For instance, using DeepSeek V3.2, DIO-Agent achieves 71.43% on the Algorithm level, a substantial jump from 17.00% via Direct prompting and 60.00% of AlphaEvolve. For Qwen-3.6-Plus, DIO-Agent leads AlphaEvolve by over 14% in the Algorithm level while maintaining a lower token cost per iteration (7068.3 vs 9274.4). Cross-model comparisons further highlight complementary strengths. These results confirm that DIO-Agent provides a model-agnostic performance boost, pushing the ceiling of automated program discovery across diverse LLM backbones.

4.3 Ablation Study

To understand the individual contribution of each component, we conduct an ablation study and report the average performance across all difficulty levels in Table 3 (comprehensive results can be found in Appendix D). The most critical component is Curriculum-wise Evolution (CE). Removing it results in the largest performance drop, with the average performance falling from 58.63% to 51.33%. Furthermore, its absence increases the token cost per iteration to 4244.31, the highest among all variants, suggesting that the curriculum not only improves accuracy but also makes the search more efficient by pruning unproductive paths early. TPP also plays a vital role, as removing it leads to a noticeable decrease in average performance from 58.63% to 53.80%. Error-Grounded Feedback (EF) serves as a refinement tool. Without it, the average performance drops to 54.61%, confirming its contribution to the overall search process.

Table 3: Ablation Study, where each module is removed separately.

Method	Performance	Token/iter
DIO-Agent	**58.63**	3738.31
w/o EF	54.61	3837.61
w/o TPP	53.80	**3369.48**
w/o CE	51.33	4244.31

4.4 Comparison with Test-Time Scaling Methods.

To demonstrate that the superiority of DIO-Agent stems from its iterative discovery logic rather than a mere increase in sampling budget, we compare it against common test-time scaling baselines: Direct, Best-of-N, and Self-Consistency. For a fair comparison, the sampling budget for these baselines is set to N=40, matching the cumulative candidate generation effort of our 40-iteration search. The results in Figure 3 reveal a massive performance gap between parallel sampling and iterative

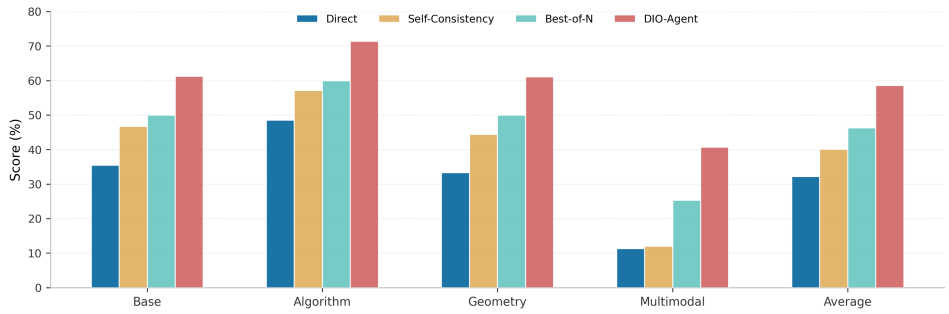
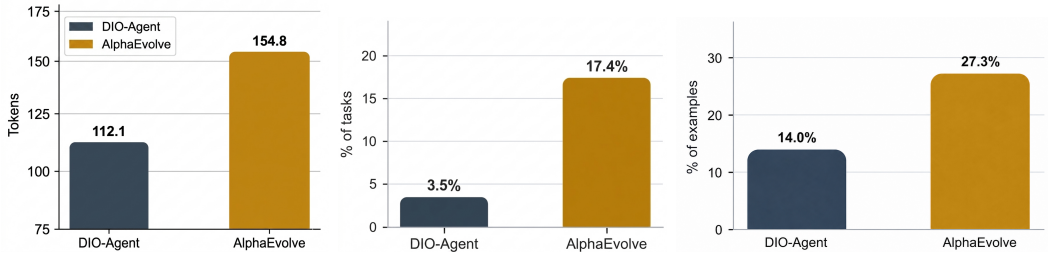


Figure 3: Comparison of DIO-Agent with test-time scaling methods.

search. While Best-of-N is the strongest scaling baseline with an average pass rate of 40.28%, it still falls nearly 24 percentage points short of DIO-Agent (63.66%), suggesting that complex problems require systematic refinement rather than repeated independent sampling. Furthermore, Direct and Self-Consistency perform significantly worse, with average pass rates of only 14.56% and 25.53%, respectively. This highlights that for program discovery tasks, the correct rule is often not the most frequent one generated by the model's prior distribution. Thus, simply increasing independent attempts is an inefficient strategy compared to the structured, feedback-driven search of DIO-Agent.



(a) Correct Code Length (b) Incorrect Early Stop (c) Copy Frequency

Figure 4: Comparison between DIO-Agent and AlphaEvolve. (a) DIO-Agent produces shorter correct programs. (b) DIO-Agent reduces incorrect early stopping during search. (c) DIO-Agent copies fewer prompt example cases into the generated programs.

4.5 Further Analysis

To better understand the underlying mechanisms of DIO-Agent, we analyze the evolutionary trajectories and behavioral patterns of the synthesized programs across multiple stages. We provide detailed numerical evidence for these analyses in Appendix E.

Evolutionary Trajectories and Program Complexity. We observe that program complexity trends are strong indicators of success. Specifically, programs that maintain a stable structure exhibit a success rate of 83.8%, typically representing simple tasks where the rule is identified early. In contrast, trajectories characterized by continuous simplification achieve a high success rate of 94.4%; after normalizing for task difficulty, this simplification signal emerges as a robust indicator of genuine discovery. While a complex-then-simple trajectory also shows a positive signal (66.7% success rate), representing an exploratory path for difficult tasks, the poorest outcomes are associated with monotonic complexity growth. The latter typically reflects patch-work programming, where the model overfits visible examples by stacking logic rather than finding a generalizable rule. Detailed trajectory statistics and the difficulty-adjusted analysis are provided in Table 7 and Table 8.

Generalization vs. Overfitting. Achieving a perfect score on visible examples does not necessarily imply the discovery of the underlying logic. Our data shows that 42 runs failed the held-out test cases; among them, 36 achieved a 1.0 accuracy on the final curriculum stage. These failed programs were significantly more verbose, with an average length of 833.9 characters, compared to the 249.8 characters of successful runs. This phenomenon is particularly prevalent in Arithmetic tasks, where

models often produce increasingly long, convoluted programs that achieve perfect training scores but collapse upon evaluation against held-out data. Genuine discovery is thus characterized by brevity and cross-stage stability rather than mere performance on visible examples. We provide the corresponding visible-stage overfitting statistics and task-family breakdown in Tables 9 and 10.

Statistic Metric. We jointly analyze program compactness, premature stopping, and copy behavior in Figure 4. Specifically, 1) As shown in Figure 4(a), DIO-Agent produces substantially shorter correct programs than AlphaEvolve (112.1 vs. 154.8 tokens), indicating reduced accumulation of redundant logic and more concise rule discovery. 2) From a search perspective, Figure 4(b) shows that only 3.5% of DIO-Agent tasks terminate early while still remaining unsolved, compared to 17.4% for AlphaEvolve. This suggests that DIO-Agent is less prone to premature convergence and more likely to continue exploration until a valid rule is discovered. 3) Figure 4(c) measures the proportion of prompt example cases that are directly copied into the generated program. DIO-Agent exhibits a substantially lower copy ratio than AlphaEvolve (14.0% vs. 27.3%), indicating that it is less prone to memorizing visible examples through explicit case replication and instead more effectively captures the underlying transformation rules. In summary, these results suggest that DIO-Agent promotes concise and generalizable program discovery while reducing both premature stopping and case-specific memorization.

4.6 Autonomous Discovery via Active Exploration

In this section, we transition from a passive evaluation setting to an Active Exploration paradigm. In this configuration, the model is no longer provided with fixed input-output pairs. Instead, it is given access to an encapsulated Ground Truth oracle. The model must autonomously design its own input cases, query the oracle for the corresponding outputs, and construct a curriculum of I/O examples to identify the underlying patterns. To evaluate the feasibility of this setting, we selected a representative subset comprising 25% of tasks from each level. This autonomous discovery mode significantly extends the search horizon for each task, as the model must not only synthesize code but also strategically select informative inputs that resolve logical ambiguities.

Table 4: Performance comparison under Autonomous Discovery setting

Methods	Base	Algorithm	Geometry	Multimodal	Average
AlphaEvolve	38.67	64.44	60.00	23.33	46.61
DIO-Agent	**52.00**	**76.67**	**64.67**	**50.00**	**60.84**

The results in Table 4 demonstrate that our DIO-Agent significantly outperforms the AlphaEvolve baseline in this interactive setting, achieving an average pass rate of 60.84% compared to 46.61%. This suggests that when given the freedom to construct its own testing environment, the agent can more effectively probe complex algorithmic boundaries than when it is restricted to a static set of provided examples. Furthermore, DIO-Agent shows remarkable robustness in the Multimodal level. This result indicates that active exploration allows the agent to systematically interrogate the multimodal LLM tool, refining its understanding of image-to-numerical mappings through targeted trial and error. Overall, the superior performance across all levels validates that autonomous input generation acts as a powerful catalyst for program discovery, enabling the agent to resolve ambiguities and verify hypotheses more rigorously than passive learning methods.

5 Related Work

Programming by Example (PBE). Programming by Example aims to synthesize programs from input-output examples. Classical PBE systems typically define a domain-specific language, search for programs that satisfy the provided examples, and rank candidate programs to resolve ambiguity [Gulwani, 2016]. Representative systems include FlashFill for spreadsheet string transformations [Gulwani, 2011], Microsoft PROSE for building DSL-based PBE tools [Polozov and Gulwani, 2015], and later frameworks that reduce the engineering cost of constructing domain-specific synthesizers [Wu et al., 2023]. With the rise of large language models, recent work revisits PBE by using LLMs as program generators, search policies, or repair agents. These studies show that LLMs can improve PBE performance when combined with synthetic training data [Li and Ellis,

2024],table-transformation prompting and hybrid solving [Zhang and Park, 2025], execution-guided within-prompt search [Verbruggen et al., 2025], compositional decomposition [Khan et al., 2025], and iterative example-based evaluation [Fu et al., 2025]. However, existing LLM-PBE work is still mostly centered on traditional PBE domains such as strings, lists, tables, and symbolic rewrite rules, leaving algorithmic reasoning, spatial reasoning, and higher-level executable rule discovery relatively underexplored.

Evolutionary Algorithms and LLM-guided Program Evolution. Evolutionary algorithms search over candidate solutions through selection, mutation, and recombination, and have developed from classical genetic algorithms to genetic programming and neuroevolution [Sampson, 1976, Koza, 1994, Stanley and Miikkulainen, 2002]. Recent systems combine this search paradigm with large language models by treating LLMs as semantic mutation operators over code. *FunSearch* first demonstrated that LLM-generated programs can be evolved with automatic evaluators to discover new mathematical constructions [Romera-Paredes et al., 2024], while *AlphaEvolve* further generalized this idea into an evolutionary coding agent that edits seed programs, executes candidates, scores them with user-defined evaluators, and selects promising variants for further improvement [Novikov et al., 2025]. Follow-up work has mainly developed in two directions: applying AlphaEvolve-style loops to new domains such as theoretical computer science, compiler optimization, and multi-agent learning [Nagda et al., 2025, Chen et al., 2026, Li et al., 2026], or improving the evolutionary procedure itself through island-based search, modular improvement and perturbation, or meta-evolution [Assumpção et al., 2025, Kravatskiy et al., 2026, Liu et al., 2026c]. However, these works typically start from a known seed program and evolve it toward a predefined objective. Inferring unknown behavioral rules directly from raw observations remains largely unexplored.

6 Conclusion

In this paper, we argued that IO2Code is an important but unexplored task different from the mainstream NL2Code task, where NL2Code evaluates the composition and reuse ability of LLMs, while IO2Code evaluates the induction and discovery ability. To this end, we established an IO2CodeBench spanning multiple difficulty levels and proposed DIO-Agent, a discovery agent that tackles IO2Code through structured evolution guided by Transformation Priority Premise. Extensive experiments show that DIO-Agent consistently outperforms existing program-by-example and evolution-agent baselines across all difficulty levels and various LLMs, while substantially surpassing test-time scaling strategies with equivalent sampling budgets. We hope that our work draws attention to the broader problem of program discovery from I/O behavior and inspires further exploration beyond natural-language-driven code generation.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *eprint arXiv: 2107.03374*, 2021.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models. *ACM Trans. Softw. Eng. Methodol.*, 33(7):182:1–182:30, 2024.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *ACM Trans. Softw. Eng. Methodol.*, 33(7):189:1–189:38, 2024.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, 2018. doi: 10.18653/v1/D18-1192.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark

dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2021.

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

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2021.

Bruce Collie, Jackson Woodruff, and Michael F. P. O'Boyle. Modeling black-box components with probabilistic synthesis. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE '20, page 1–14. ACM, November 2020. doi: 10.1145/3425898.3426952.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. Large language models meet nl2code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, pages 7443–7464, 2023.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020*, pages 1536–1547, 2020.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 conference on empirical methods in natural language processing*, pages 8696–8708, 2021.

Robert C Martin. *Clean Craftsmanship: Disciplines, Standards, and Ethics*. Addison-Wesley Professional, 2021.

Sumit Gulwani. Programming by examples. *Dependable Software Systems Engineering*, 45(137): 3–15, 2016.

Gust Verbruggen, Ashish Tiwari, Mukul Singh, Vu Le, and Sumit Gulwani. Execution-guided within-prompt search for programming-by-example. In *The Thirteenth International Conference on Learning Representations*, 2025.

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

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

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

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126, 2015.

Jiarong Wu, Lili Wei, Yanyan Jiang, Shing-Chi Cheung, Luyao Ren, and Chang Xu. Programming by example made easy. *ACM Transactions on Software Engineering and Methodology*, 33(1): 1–36, 2023.

Wen-Ding Li and Kevin Ellis. Is programming by example solved by llms? *Advances in Neural Information Processing Systems*, 37:44761–44790, 2024.

Shuning Zhang and Yongjoo Park. Pbe meets llm: When few examples aren't few-shot enough. *arXiv preprint arXiv:2507.05403*, 2025.

Ruhma Khan, Sumit Gulwani, Vu Le, Arjun Radhakrishna, Ashish Tiwari, and Gust Verbruggen. Llm-guided compositional program synthesis. *arXiv preprint arXiv:2503.15540*, 2025.

Yingjie Fu, Bozhou Li, Linyi Li, Wentao Zhang, and Tao Xie. The first prompt counts the most! an evaluation of large language models on iterative example-based code generation. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1583–1606, 2025.

Jeffrey R Sampson. Adaptation in natural and artificial systems (john h. holland). *SIAM Review*, 18 (3):529–530, 1976.

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

Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

Mengyang Liu, Taozhi Chen, Zhenhua Xu, Xue Jiang, and Yihong Dong. Multi-agent collaboration with state management. *arXiv preprint arXiv:2605.20563*, 2026a. URL `https://arxiv.org/abs/2605.20563`.

Chun-Wah Hsu, Kai Gong, Yu Wu, Xianhe Chen, Mengyang Liu, Jie Li, Hanyu Li, Zhixuan Liu, Naisheng Tang, Jiaying Chi, Ziheng Fan, Xuning He, Xiaokang Yang, Xue Jiang, and Yihong Dong. OpenCollab: A multi-agent coding framework with programmable collaboration and controllable runtime. *arXiv preprint arXiv:2609.38345*, 2026. URL `https://arxiv.org/abs/2609.38345`.

Mengyang Liu, Weiming Wang, Haixin Nan, Yu Wu, Peize Li, Shihong Huang, Xuning He, Xue Jiang, and Yihong Dong. SPROUT: Scaling agentic reinforcement learning for long-horizon software engineering. *arXiv preprint arXiv:2610.14405*, 2026b. URL `https://arxiv.org/abs/2610.14405`.

Xuning He, Kai Fan, Runyu Zhang, Binghuan Wang, Yuxiang Zhang, Xue Jiang, and Yihong Dong. In-model context compression with agentic soft tokens for long-horizon LLM agents. *arXiv preprint arXiv:2610.41291*, 2026. URL `https://arxiv.org/abs/2610.41291`.

Ansh Nagda, Prabhakar Raghavan, and Abhradeep Thakurta. Reinforced generation of combinatorial structures: Applications to complexity theory. *arXiv e-prints*, pages arXiv–2509, 2025.

Hongzheng Chen, Alexander Novikov, Ngân Vũ, Hanna Alam, Zhiru Zhang, Aiden Grossman, Mircea Trofin, and Amir Yazdanbakhsh. Magellan: Autonomous discovery of novel compiler optimization heuristics with alphaevolve. *arXiv preprint arXiv:2601.21096*, 2026.

Zun Li, John Schultz, Daniel Hennes, and Marc Lanctot. Discovering multiagent learning algorithms with large language models. *arXiv preprint arXiv:2602.16928*, 2026.

Alexey Kravatskiy, Valentin Khrulkov, and Ivan Oseledets. Improvevolve: Ask alphaevolve to improve the input solution and then improvise. *arXiv preprint arXiv:2602.10233*, 2026.

Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z Pan, et al. Evox: Meta-evolution for automated discovery. *arXiv preprint arXiv:2602.23413*, 2026c.

Eric J Michaud, Isaac Liao, Vedang Lad, Ziming Liu, Anish Mudide, Chloe Loughridge, Zifan Carl Guo, Tara Rezaei Kheirkhah, Mateja Vukelić, and Max Tegmark. Opening the ai black box: program synthesis via mechanistic interpretability. *arXiv preprint arXiv:2402.05110*, 2024.

Qwen Team. Qwen3.5: Towards native multimodal agents. `https://qwen.ai/blog?id=qwen3.5`, 2026. Accessed: 2026-05-07.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, et al. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. Deepseek-v3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

A Stage-wise Analysis of Curriculum Refinement

We distinguish between the strict task-level success metric used for final evaluation and a diagnostic sample-level pass ratio used only for analyzing the search process. In the final evaluation, we follow the strict all-pass criterion: a task is considered solved only if the generated program passes all held-out tests for that task. If any held-out test case fails, the task is counted as unsolved. We conduct a stratified random sampling of over 30 non-multimodal tasks according to task categories. Under the strict all-pass criterion, DIO-Agent solves 15 out of 30 tasks, while AlphaEvolve solves 11 out of 30 tasks.

To better understand how the curriculum affects the search process before a task is fully solved, we additionally report a diagnostic metric: the sample-level pass ratio under the full evaluator. This metric measures the fraction of held-out test cases passed by the current best program for each task. It is not used to determine whether a task is solved; instead, it allows us to observe partial progress when a program has not yet passed all held-out tests.

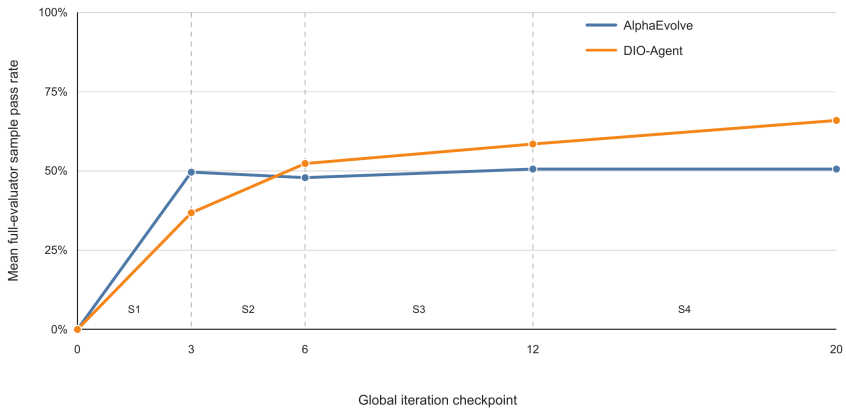


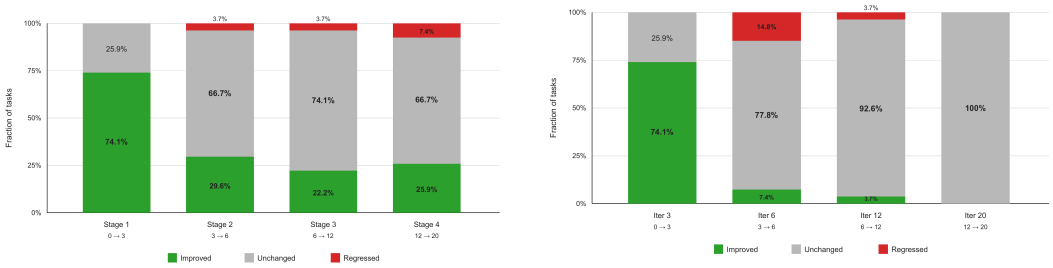
Figure 5: Mean full-evaluator sample pass rate across global iteration checkpoints. DIO-Agent starts below AlphaEvolve at the first checkpoint, but its performance improves steadily as the curriculum progresses. In contrast, AlphaEvolve remains nearly flat after the early checkpoint.

As shown in Figure 5, DIO-Agent exhibits a clear upward trajectory across curriculum stages. Its mean full-evaluator sample pass ratio increases from 36.8% after stage 1 to 52.3% after stage 2, 58.5% after stage 3, and finally 65.9% after stage 4. In contrast, AlphaEvolve remains around 50% under the same checkpoints, with sample pass ratios of 49.6%, 47.9%, 50.6%, and 50.6%, respectively. Therefore, although DIO-Agent is initially lower than AlphaEvolve at the first checkpoint, it continues to benefit from later curriculum stages and eventually surpasses AlphaEvolve.

We further analyze how each checkpoint changes the per-task sample-level pass ratio. A task is counted as improved if its sample pass ratio increases compared with the previous checkpoint, and counted as regressed if its sample pass ratio decreases. This analysis helps separate two effects: whether the overall average performance improves, and how many individual tasks receive additional gains at each stage.

For DIO-Agent, stage 1 improves 22 out of 30 tasks and does not regress any task, suggesting that the early curriculum helps the agent quickly form useful initial hypotheses from simple examples. In later stages, the number of additionally improved tasks becomes smaller: 8, 6, and 7 tasks improve after stages 2, 3, and 4, respectively. Meanwhile, only a few tasks regress, with 1, 1, and 2 regressions in stages 2–4. This does not mean that DIO-Agent becomes worse in later stages, since its mean sample pass ratio continues to increase. Instead, it indicates that the marginal gain of later stages becomes smaller. The role of the curriculum gradually shifts from making many tasks immediately better to validating and refining the hypotheses discovered in earlier stages using more diverse and more difficult examples.

AlphaEvolve shows a different pattern. Its improvement is concentrated at the early checkpoint: after iteration 3, it improves 21 out of 30 tasks. However, its later improvements quickly stagnate. After iterations 6, 12, and 20, only 2, 1, and 0 tasks continue to improve, respectively, and iteration



(a) DIO-Agent (b) AlphaEvolve

Figure 6: Per-checkpoint task-level changes measured by the diagnostic sample-level pass ratio. Green indicates tasks whose sample pass ratio improves compared with the previous checkpoint, gray indicates unchanged tasks, and red indicates regressed tasks.

6 even causes 4 tasks to regress. This contrast suggests that DIO-Agent's curriculum stages do more than simply increase the number of search iterations. By progressively expanding the visible examples, the curriculum provides continued validation pressure, which helps the search process filter out unstable programs and refine promising hypotheses.

B Comprehensive results

Table 5: Comprehensive results of per-task pass ratios on training and test examples, completed with additional baselines.

Category	Task	Set	Direct	PBE	E-PBE	TDD	FunSearch	AlphaEvolve	CodeEvolve	DIO-Agent
Base	Abs_Current	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Abs_Diff	Train	0.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.00	0.00	1.00	0.87	0.93	1.00	1.00	1.00
	Add_Mod_3	Train	0.00	0.00	0.00	1.00	0.00	0.00	1.00	1.00
		Test	0.00	0.00	0.00	0.87	0.00	0.00	0.93	1.00
	Add_Mod_4	Train	0.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.00	0.00	0.00	0.87	0.00	0.00	0.93	1.00
	Add_Mod_5	Train	0.00	0.00	1.00	0.25	1.00	1.00	0.25	0.25
		Test	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Add_Mod_6	Train	0.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Add_Mod_7	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.67	0.60	0.60	0.87	0.67	0.67	1.00	1.00
	Add_Mod_8	Train	0.75	0.75	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.27	0.20	1.00	0.87	0.93	1.00	1.00	1.00
	Alternating_Last3	Train	0.38	0.38	0.38	0.75	0.38	0.38	0.75	0.75
		Test	0.27	0.20	0.06	0.00	0.13	0.13	0.00	0.07
	Alternating_Last4	Train	0.25	0.25	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.07	0.00	0.60	0.34	0.67	0.67	0.40	0.47
	Balanced_Parenthesis	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.80	0.73	0.73	0.67	0.80	0.80	1.00	0.80
	Base_3_Addition	Train	0.00	0.00	0.25	1.00	0.27	0.25	1.00	1.00
		Test	0.00	0.00	0.20	0.07	0.27	0.27	0.13	0.20
	Base_4_Addition	Train	0.50	0.50	0.38	1.00	0.38	0.38	1.00	1.00
		Test	0.00	0.00	0.06	0.00	0.13	0.13	0.06	0.13
	Base_5_Addition	Train	0.25	0.25	0.25	0.88	0.25	0.25	0.88	0.88
		Test	0.00	0.00	0.06	0.00	0.13	0.13	0.00	0.00
	Base_6_Addition	Train	0.00	0.00	0.25	1.00	0.25	0.25	1.00	1.00
		Test	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.07
	Base_7_Addition	Train	0.13	0.13	0.38	1.00	0.38	0.38	1.00	1.00
		Test	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Binary_Addition	Train	0.50	0.50	0.50	1.00	0.50	0.50	1.00	1.00
		Test	0.20	0.13	0.20	0.60	0.27	0.27	0.66	0.73
	Bit_Dot_Prod_Mod2	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.47	0.40	0.40	0.34	0.47	0.47	0.40	0.47
	Bit_Palindrome	Train	0.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.07
	Bit_Shift_Right	Train	0.63	0.63	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.27	0.20	1.00	0.87	0.93	1.00	1.00	1.00
	Bitwise_And	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	0.87	1.00	0.33	1.00	1.00
	Bitwise_Not	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Bitwise_Or	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Bitwise_Xor	Train	0.00	0.13	1.00	1.00	1.00	1.00	1.00	1.00

Continued on next page

Category	Task	Set	Direct	PBE	E-PBE	TDD	FunSearch	AlphaEvolve	CodeEvolve	DIO-Agent
		Test	0.20	0.13	1.00	0.87	1.00	1.00	1.00	1.00
	Current_Number	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Diff_Abs_Values	Train	0.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.00	0.00	0.93	0.87	0.93	1.00	1.00	1.00
	Diff_Last2	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Dithering	Train	0.00	0.00	0.13	0.38	0.20	0.13	0.38	0.38
		Test	0.00	0.00	0.13	0.00	0.20	0.20	0.00	0.00
	Div_3	Train	0.00	0.00	0.38	0.50	0.38	0.38	0.50	0.50
		Test	0.00	0.00	0.00	0.00	0.07	0.07	0.00	0.00
	Div_5	Train	0.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.00	0.00	0.13	0.00	0.20	0.20	0.00	0.00
	Div_7	Train	0.00	0.00	1.00	0.50	1.00	1.00	0.50	0.50
		Test	0.00	0.00	0.26	0.14	0.33	0.33	0.20	0.27
	Evens_Counter	Train	0.00	0.00	0.25	1.00	0.25	0.25	1.00	1.00
		Test	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Evens_Detector	Train	0.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.00	0.00	0.93	0.87	0.93	1.00	1.00	1.00
	Majority_0_1	Train	0.00	0.00	0.63	1.00	0.63	0.63	1.00	1.00
		Test	0.00	0.00	0.33	0.00	0.40	0.40	0.06	0.13
	Majority_0_2	Train	0.50	0.50	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.40	0.33	0.46	0.47	0.53	0.53	0.53	0.60
	Majority_0_3	Train	0.88	0.88	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.80	0.73	0.66	0.74	0.73	0.73	1.00	0.87
	Max_Seen	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	1.00	1.00	0.87	1.00	1.00	1.00	1.00
	Min_Seen	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	1.00	1.00	0.87	1.00	1.00	1.00	1.00
	Newton_Freebody	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Newton_Gravity	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Newton_Magnetic	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Newton_Spring	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Parity_All	Train	0.50	0.50	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.20	0.13	1.00	0.87	1.00	1.00	1.00	1.00
	Parity_Bits_Mod2	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	0.87	1.00	1.00	1.00	1.00
	Parity_Last2	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	1.00	1.00	0.87	1.00	1.00	1.00	1.00
	Parity_Last3	Train	0.00	0.00	0.50	1.00	0.50	0.50	1.00	1.00
		Test	0.00	0.00	0.20	0.87	0.27	0.27	1.00	1.00
	Parity_Last4	Train	0.63	0.63	0.63	0.75	0.63	0.63	0.75	0.75
		Test	0.40	0.33	0.33	0.20	0.40	0.40	0.26	0.33
	Parity_Zeros	Train	0.13	0.13	1.00	0.25	1.00	1.00	0.25	0.25
		Test	0.20	0.13	0.00	0.00	0.00	0.00	0.00	0.00
	Perfect_Square_Detector	Train	0.88	0.88	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.20	0.13	0.13	0.87	0.20	0.20	0.93	1.00
	Prev1	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Prev2	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	1.00	1.00	1.00	1.00	1.00
	Prev3	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	1.00	1.00	1.00	1.00	1.00
	Prev4	Train	0.25	0.25	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.20	0.13	1.00	0.67	1.00	1.00	1.00	0.80
	Prev5	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	1.00	1.00	1.00	1.00	1.00
	Previous_Equals_Current	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	0.87	1.00	1.00	1.00	1.00
	Sum_All	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	0.87	1.00	1.00	1.00	1.00
	Sum_Last2	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	0.87	1.00	1.00	1.00	1.00
	Sum_Last3	Train	0.25	0.25	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.13	0.06	1.00	0.87	1.00	1.00	1.00	1.00
	Sum_Last4	Train	0.25	0.25	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.27	0.20	1.00	0.87	1.00	1.00	1.00	1.00
	Sum_Last5	Train	0.75	0.75	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.40	0.33	1.00	0.87	1.00	1.00	1.00	1.00
	Sum_Last6	Train	0.88	0.88	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.53	0.46	1.00	0.87	1.00	1.00	1.00	1.00
	Sum_Last7	Train	0.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.00	0.00	1.00	0.87	1.00	0.60	1.00	1.00
Algorithm	AllPositive	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.80	0.73	0.73	0.87	1.00	0.80	0.93	1.00
	BalancedParentheses	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.47	0.40	0.40	0.40	0.47	0.47	0.46	0.53
	BinaryAddition	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	1.00	1.00	1.00	1.00	1.00
	ClimbStairs	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.93	0.86	1.00	0.87	1.00	1.00	1.00	1.00
	CountChars	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	CountDigits	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Continued on next page

Category	Task	Set	Direct	PBE	E-PBE	TDD	FunSearch	AlphaEvolve	CodeEvolve	DIO-Agent
		Test	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	CountZeros	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Dec2Bin	Train	0.00	0.13	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.20	0.13	1.00	0.87	1.00	1.00	0.93	1.00
	Dec2Roman	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	1.00	1.00	1.00	1.00	1.00
	Factorial	Train	0.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.00	0.00	0.93	1.00	1.00	1.00	0.93	1.00
	Fibonacci	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	1.00	1.00	1.00	1.00	1.00
	GCD	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	HammingWeight	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	1.00	1.00	1.00	1.00	1.00
	HouseRobber	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	0.87	1.00	0.87	1.00	1.00
	IntegerDivision	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	1.00	1.00	1.00	1.00	1.00
	IsPalindrome	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.47	0.40	0.40	0.87	0.47	0.47	0.93	1.00
	IsSorted	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.73	0.66	0.66	0.60	0.73	0.73	0.66	0.73
	KthRoot	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.87	0.80	0.20	0.14	0.27	0.27	0.20	0.27
	KthRootMultidigit	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.93	0.86	0.66	0.40	0.73	0.73	0.46	0.53
	LeadingZeros	Train	0.88	0.88	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.40	0.33	0.73	0.87	0.80	0.80	0.93	1.00
	ListProduct	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.93	0.86	1.00	1.00	1.00	1.00	1.00	1.00
	ListSum	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	LongestIncreasingSubseqLen	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	0.46	0.40	0.53	0.53	0.46	0.53
	Modulo	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Multiply	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	PolishRPN	Train	0.50	0.60	0.88	0.75	0.88	0.88	0.75	0.75
		Test	0.67	0.60	0.66	0.60	0.73	0.73	0.66	0.73
	PolishRPN2	Train	0.50	0.66	0.66	0.63	0.73	0.50	0.63	0.63
		Test	0.73	0.66	0.66	0.07	0.73	0.73	0.13	0.20
	PolishRPN3	Train	0.00	0.06	0.75	0.50	0.75	0.75	0.50	0.50
		Test	0.13	0.06	0.00	0.00	0.07	0.07	0.00	0.07
	Power	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	1.00	1.00	1.00	1.00	1.00
	ReverseList	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	1.00	1.00	1.00	1.00	1.00
	ReverseNumber	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	1.00	1.00	1.00	1.00	1.00
	Sign	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	0.87	1.00	1.00	1.00	1.00
	SparseParity	Train	0.88	0.88	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.47	0.40	0.33	0.47	0.40	0.40	1.00	0.60
	SumDigits	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	0.87	1.00	1.00	1.00	1.00
	TrailingZeros	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.80	0.73	0.73	0.67	0.80	0.80	0.73	0.80
Geometry	CheckStraightLine	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.80	0.73	0.73	1.00	0.80	0.80	0.66	0.73
	MinTimeVisitPoints	Train	0.50	0.50	0.50	1.00	0.50	0.50	1.00	1.00
		Test	0.07	0.00	0.13	0.00	0.20	0.20	0.06	0.13
	MaxPointsOnLine	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.80	0.73	1.00	1.00	1.00	1.00	0.93	1.00
	NumberOfBoomerangs	Train	0.88	0.88	0.38	0.50	0.38	0.38	0.50	0.50
		Test	0.53	0.46	0.26	0.40	0.33	0.33	0.46	0.53
	ValidSquare	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	0.93	1.00	1.00	1.00	1.00	1.00
	LargestTriangleArea2	Train	0.13	0.13	0.26	1.00	0.33	0.25	1.00	1.00
		Test	0.00	0.00	0.26	0.40	0.33	0.33	0.46	0.53
	BoundingBoxArea	Train	0.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.00	0.00	0.93	1.00	1.00	1.00	1.00	1.00
	BoundingBoxPerimeter	Train	0.38	0.53	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.60	0.53	0.93	1.00	1.00	1.00	1.00	1.00
	ChebyshevDistance	Train	0.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.00	0.00	0.93	1.00	0.93	1.00	1.00	1.00
	EuclideanDistanceSquared	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	1.00	1.00	1.00	1.00	1.00
	HasDuplicatePoints	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	0.86	0.87	0.93	0.93	0.93	1.00
	IsAxisAligned	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.93	0.86	0.86	1.00	0.93	0.93	0.86	0.93
	ManhattanDistance	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	1.00	1.00	1.00	1.00	1.00
	MidpointIntegerCheck	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.80	0.73	0.73	0.67	0.80	0.80	0.73	0.80
	Orientation	Train	0.75	0.75	1.00	1.00	1.00	1.00	1.00	1.00

Continued on next page

Category	Task	Set	Direct	PBE	E-PBE	TDD	FunSearch	AlphaEvolve	CodeEvolve	DIO-Agent
		Test	0.40	0.33	0.20	0.40	0.27	0.27	0.46	0.53
	PathReturnsOrigin	Train	0.88	0.88	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.93	0.86	0.93	1.00	0.93	1.00	0.93	1.00
	RectangleAreaByDiagonal	Train	0.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	0.00	0.00	0.93	0.87	0.93	1.00	1.00	1.00
	TriangleArea2	Train	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Test	1.00	0.93	1.00	1.00	1.00	1.00	1.00	1.00
Multimodal	CellCount	Train	–	0.53	0.60	0.67	0.73	0.75	0.73	0.88
		Test	–	0.40	0.47	0.47	0.60	0.60	0.60	0.60
	DiskCount	Train	–	0.80	0.93	0.87	1.00	1.00	1.00	1.00
		Test	–	0.67	0.80	0.67	1.00	1.00	1.00	1.00
	MaxRadius	Train	–	0.20	0.27	0.33	0.40	0.25	0.47	0.63
		Test	–	0.07	0.13	0.13	0.27	0.13	0.33	0.47
	PeakCount	Train	–	0.27	0.27	0.40	0.47	0.25	0.53	0.63
		Test	–	0.13	0.13	0.20	0.33	0.20	0.40	0.53
	TopIndex	Train	–	0.20	0.20	0.27	0.27	0.25	0.27	0.50
		Test	–	0.07	0.07	0.07	0.13	0.00	0.13	0.20
	UpperCount	Train	–	0.13	0.13	0.20	0.20	0.00	0.20	0.25
		Test	–	0.00	0.00	0.00	0.07	0.00	0.07	0.00
	PolySides	Train	–	0.27	0.27	0.40	0.33	0.00	0.53	0.75
		Test	–	0.13	0.13	0.20	0.20	0.00	0.40	0.73
	PieMax	Train	–	0.27	0.27	0.27	0.27	0.38	0.27	0.38
		Test	–	0.13	0.13	0.07	0.13	0.20	0.13	0.20
	ArrowDir	Train	–	0.20	0.20	0.27	0.20	0.38	0.20	0.63
		Test	–	0.07	0.07	0.07	0.07	0.00	0.07	0.07
	RadarMax	Train	–	0.20	0.27	0.27	0.20	0.38	0.27	0.50
		Test	–	0.07	0.13	0.07	0.07	0.07	0.13	0.27

C Hyperparameter Impact

We further investigate the sensitivity of DIO-Agent to two key hyperparameters: the number of independent evolutionary islands and the total number of search iterations. These parameters represent the balance between exploration breadth and search depth.

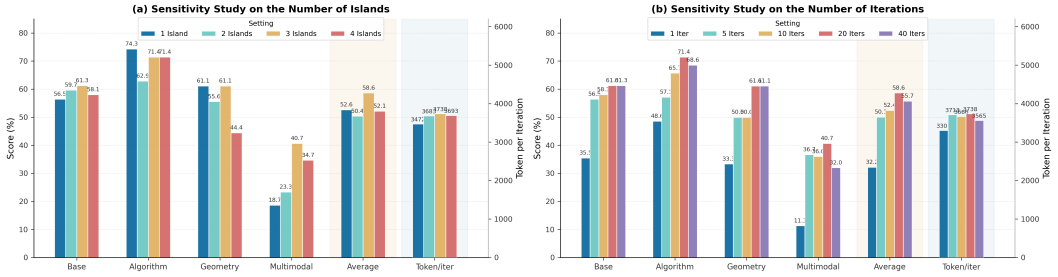


Figure 7: Sensitivity studies on different hyperparameters. (a) Sensitivity study on the number of islands. (b) Sensitivity study on the number of iterations.

Impact of Island Count. As shown in Figure 7(a), the average performance peaks when using 3 islands, achieving a pass rate of 64.61%. While a single island provides surprisingly strong results in the Algorithm level (74.29%), it struggles with the Base level, likely due to a lack of diverse candidate rules during the early stages of discovery. Increasing the count to 4 islands leads to a performance regression, particularly in the Geometry level (dropping to 44.44%). This suggests that while parallel islands encourage exploration and prevent premature convergence, an excessive number of islands may dilute the quality of the population or lead to resource fragmentation within the fixed iteration budget, thereby hindering the fine-tuning of complex geometric logic.

Impact of Search Iterations. Figure 7(b) illustrates the relationship between the number of iterations and the discovery success rate. Performance scales positively with search depth up to 20 iterations, where the average pass rate reaches its maximum of 64.61%. The improvement is most significant between 5 and 20 iterations, particularly in the Algorithm and Geometry levels, which require multiple refinement steps to identify non-obvious algorithmic patterns. Interestingly, increasing the depth further to 40 iterations does not yield additional gains; instead, we observe a slight dip in the Algorithm level performance (68.57%). This plateau indicates that 20 iterations provide a sufficient search horizon for the current benchmark, and further expansion may lead to diminishing returns or marginal overfitting on visible examples that does not translate to held-out

test cases. Notably, the token cost remains relatively stable across these variations, suggesting that the complexity of each iteration is well-contained by our framework's structured design.

D Detailed Ablation Results

To understand the individual contribution of each component, we conduct an ablation study across all difficulty levels. The results are shown in Table 6. The most critical component is Curriculum-wise Evolution (CE). Removing it results in the largest performance drop, with the average pass rate falling from 58.63% to 51.33%. This decline is particularly pronounced in the Algorithm level (from 71.43% to 62.86%). Furthermore, its absence increases the token cost per iteration to 4244.31, suggesting that the curriculum not only improves accuracy but also makes the search more efficient by pruning unproductive paths early. TPP also plays a vital role, particularly in specialized domains. Its removal leads to a substantial decrease in Geometry performance (from 61.11% to 50.00%) and a noticeable drop in Algorithm (from 71.43% to 68.57%), indicating that structured complexity ordering is crucial for tasks involving non-trivial algorithmic structures. Error-Grounded Feedback (EF) serves as a critical refinement tool. While its impact on the Algorithm level is negligible, its absence causes a sharp decline in Geometry (from 61.11% to 50.00%) and Multimodal (from 40.67% to 37.33%), suggesting that error traces are instrumental in correcting precise logical errors common in these domains.

Table 6: Ablation Study, where each module is removed separately.

Method	IO2CodeBench					
	Base	Algorithm	Geometry	Multimodal	Average	Token/iter
DIO-Agent	**61.29**	**71.43**	61.11	40.67	58.63	3738.31
w/o EF	59.68	71.43	50.00	37.33	54.61	3837.61
w/o TPP	61.29	68.57	50.00	35.33	53.80	3369.48
w/o CE	56.45	62.86	50.00	36.00	51.33	4244.31

E Detailed Behavioral Analysis

This appendix provides additional behavioral analyses of DIO-Agent on non-multimodal tasks. We exclude the Multimodal tier from these analyses because multimodal tasks involve an auxiliary LLM for image-to-value extraction, making program complexity, mutation attribution, and failure trajectories less directly comparable to purely code-based tasks.

Table 7 analyzes how the complexity of the best program evolves across curriculum stages. Successful runs are often associated with either stable complexity or monotonic simplification. In particular, the monotone-down pattern achieves the highest held-out success rate, suggesting that DIO-Agent frequently discovers a more general rule by first proposing a workable structure and then compressing unnecessary logic. In contrast, monotone-up trajectories have a much lower success rate, which is consistent with the failure mode of accumulating patches to satisfy visible examples rather than identifying the underlying transformation.

Table 7: Complexity trajectories and held-out success.

Trajectory type	Runs	Held-out successes	Success rate	Interpretation
Stable	37	31	83.8%	Rule found early; little structural change
Monotone down	18	17	94.4%	Continuous simplification
Hump: up then down	18	12	66.7%	Explore, then compress
Monotone up	34	9	26.5%	Patch accumulation / overfitting
Valley: down then up	4	3	75.0%	Rare mixed simplification pattern
Mixed	4	1	25.0%	No consistent direction
Total	115	73	63.5%	

Table 8 further controls for task difficulty. Task difficulty is estimated from non-hybrid baselines and is not computed from the focal DIO-Agent run, so the residual reflects whether a trajectory type

outperforms tasks of comparable estimated difficulty. The positive residual for monotone-down trajectories shows that their high success rate is not merely a consequence of easier tasks. Conversely, monotone-up trajectories remain strongly negative after difficulty adjustment, reinforcing the interpretation that increasing complexity across stages is a warning sign for overfitting or patch accumulation.

Table 8: Difficulty-adjusted trajectory analysis.

Trajectory type	Hybrid runs	Success rate	Avg. task difficulty	Success residual
Monotone down	173	90.8%	0.750	+0.212
Stable	203	69.0%	0.793	-0.029
Hump: up then down	148	61.5%	0.541	+0.047
Monotone up	207	24.2%	0.357	-0.228
Valley: down then up	44	65.9%	0.389	+0.216
Mixed	29	51.7%	0.363	-0.002

Table 9 distinguishes visible-set fitting from true held-out generalization. Among the held-out failures, many programs achieve perfect accuracy on the final visible curriculum stage but still fail on hidden test cases. These curriculum-perfect failures are substantially longer than held-out-successful programs, indicating that perfect visible accuracy alone is not a reliable indicator of rule discovery. Instead, excessive program length often reflects memorization or case-specific branching that does not transfer to unseen inputs.

Table 9: Visible-stage success versus held-out generalization.

Group	Runs	Visible stage-4 accuracy	Held-out outcome	Avg. code length
Held-out-successful runs	73	–	Success	249.8 chars
Held-out failures	42	–	Failure	–
Stage-4 curriculum-perfect but held-out-failed	36	1.0	Failure	833.9 chars

Table 10: Overfitting failures by task family.

Task family	Tasks	Overfit failures	Overfit rate	Avg. failed length
Arithmetic	15	9	60.0%	1092.2 chars
Core	12	6	50.0%	806.5 chars
LeetCode_Geo	18	7	38.9%	246.7 chars
Extra	35	11	31.4%	758.5 chars
BitParity	13	2	15.4%	2268.0 chars
Sequence	18	1	5.6%	743.0 chars
Newton	4	0	0.0%	–

Table 10 breaks down the curriculum-perfect but held-out-failed cases by task family. Arithmetic and Core tasks show the highest overfitting rates, suggesting that these families are especially prone to visible-set shortcuts. This is plausible because arithmetic tasks often admit many partial rules that match a small number of examples but fail under broader coverage. In contrast, Sequence and Newton tasks show much lower overfitting rates, indicating that visible examples in these families may provide stronger constraints on the underlying rule.

F Case Study

In this section, we study the performance of DIO-Agent across three specific problems. As shown in Figure 8, the first case involves a geometry task. The input consists of an indefinite number of point coordinates (greater than 3), and the required output is the area of the largest triangle formed by these points. Under our Curriculum-wise Evolution approach, the model successfully discovered the underlying relationship between the inputs and outputs. The second case explores a multimodal scenario, illustrated in Figure 9. The input is a grid image, and the output corresponds to the total number of 1×1 squares within the image. During the solution process, the model opted to call our provided multimodal model as a tool to extract specific visual information, leading to a successful

Case: Finding Geometry Rules

Task Description: Largest triangle area

Example 1:
Input: [[-13, -19], [-3, -5], [-6, -12], [-14, 14], [-15, 17], [7, -18], [-19, -15]]
Output: 844
Example 2:
Input: [[-6, 12], [18, -19], [15, -8]]
Output: 171
.....

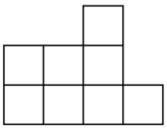
Best Solution in DIO-Agent:

```
def process(input_data):
    n = len(input_data)
    if n <= 2:
        return 0
    result = 0
    for i in range(n):
        for j in range(i+1, n):
            for k in range(j+1, n):
                x1, y1 = input_data[i]
                x2, y2 = input_data[j]
                x3, y3 = input_data[k]
                det = abs(x1*(y2-y3) + x2*(y3-y1) + x3*(y1-y2))
                if det > result:
                    result = det
    return result
```

Figure 8: LLM manages to discover the geometry rule behind the I/O examples

Case: Prompting a External Multimodal LLM

Task Example:

Input: 

Output: 8 (Number of 1*1 squares)

Best Solution in DIO-Agent:

```
def solve(input_data):
"""
 Return result for the given input_data.
"""
    image_path = input_data['image_path']

    # Use the multimodal model to directly compute the answer by providing examples

    model = os.environ.get("MM_PARSE_MODEL", "openai/gpt-4o-mini")
    api_key = os.environ.get("OPENROUTER_API_KEY")

    if api_key:
        import requests
        import base64

        # Encode the image
        withopen(image_path, "rb") as img_file:
        base64_image = base64.b64encode(img_file.read()).decode('utf-8')

        # Provide context with all training examples
        prompt ="""We have a pattern recognition task. Given training examples:
               Image 1 -> 16
               Image 2 -> 8
               Image 3 -> 9
               Image 4 -> 6
               Image 5 -> 6
               Image 6 -> 14
               Image 7 -> 14
               Image 8 -> 10
            The pattern likely involves counting the number of distinct colored
        shapes or objects in the image.Look at the image carefully, count the number
        of distinct objects (like circles, squares, etc.), and output that
        number.Return only the integer number, nothing else."""
        ......

        answer = response.json()["choices"][0]["message"]["content"].strip()
        # Try to extract integer from answer
        import re
        numbers = re.findall(r'\b\d+\b', answer)
        if numbers:
            return int(numbers[0])
        except:
            # Last resort: return a default
            return 0
```

Preliminarily Discover the mapping rule

Figure 9: LLM utilizes an external multimodal LLM as a tool to solve Multimodal problems

Case: Directly Utilizing Python Tools

Task Example:

Input:



Output: 5 (Number of distict circles)

Best Solution in DIO-Agent:

```
import cv2
import numpy as np

def solve(input_data):
    """
    Return result for the given input_data.
    """
    image_path = input_data['image_path']
    # Read image
    img = cv2.imread(image_path)
    if img is None:
        return 0

    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    # Apply Gaussian blur to reduce noise
    gray = cv2.medianBlur(gray, 5)
```

Use cv2 to analyze the image

```
    # Detect circles using HoughCircles
    circles = cv2.HoughCircles(gray, cv2.HOUGH_GRADIENT, dp=1, minDist=20,
                               param1=50, param2=30, minRadius=10, maxRadius=100)

    if circles is not None:
        circles = np.uint16(np.around(circles))
        return len(circles[0])
    else:
        return 0
```

Figure 10: LLM directly utilizes python packages as tools to solve Multimodal problems

answer. The third case is also a multimodal scenario, shown in Figure 10. The input image contains circles of various colors (where the color variations serve as distractors), and the output is the total count of circles. In this instance, the LLM chose to directly utilize the cv2 package, calling specific functions for information extraction to achieve the final solution. These experiments demonstrate that DIO-Agent can learn to diversify its problem-solving strategies based on the specific requirements of different task types.

G Limitation

Our work has two main limitations. First, although the proposed IO2CodeBench covers multiple difficulty levels ranging from basic data operations to complex algorithmic reasoning (e.g., geometric computation and multimodal tasks), the landscape of code generation includes numerous specialized fields yet to be fully explored. Expanding the domain coverage would enable a more comprehensive evaluation of different methods, and this constitutes an important direction for our future work. Second, the current experimental setup uniformly provides fixed visible I/O examples per task for search. However, in real-world applications, the number of available I/O examples may vary considerably, and different examples can differ significantly in their informativeness for revealing the underlying rule. Certain well-chosen boundary cases may be far more discriminative than a large number of routine examples. The impact of the quantity, diversity, and informational coverage of I/O examples on search effectiveness warrants further systematic investigation.

H IO2CodeBench

This appendix describes the construction of the IO2CodeBench used in our experiments. The benchmark is programmatically generated from deterministic oracle functions, therefore every example pair is produced automatically rather than annotated by humans.

Prompt

```
You are an expert programmer specializing in program synthesis from examples.

 Your task is to implement a function that works correctly on the given input-output examples.
 You need to infer the pattern from these examples and write code that generalizes to all test cases.

 **Training Examples (you can see these):**
 Example 1:
 Input: ......
 Output: ......

 **IMPORTANT NOTES:**
 - These are TRAINING examples only (randomly generated with seed=42)
 - Your program will be evaluated on a SEPARATE TEST SET (seed=999) that you haven't seen
 - You must infer the GENERAL PATTERN, not memorize specific examples
 - The test set has different inputs than the training examples shown above

 **Requirements:**
 - Input: list[int]
 - Output: number
 - The function must generalize to unseen examples
 - Don't hardcode mappings for specific inputs
 - Handle edge cases appropriately (empty lists, single elements, etc.)

 **Your goal:**
 Write a function that achieves 100% accuracy on the UNSEEN test set by learning the pattern from
training examples.

### TPP priority
1) nil -> constant
2) constant -> scalar
3)statement -> statements
4) unconditional -> if
5) scalar -> array
6) if -> while
7) expression-> function

### Anti-hardcode rules
- Do NOT match full concrete inputs(e.g. exact list equality checks).
- Do NOT build dictionary lookups from seen examples.
- Infer a general rule that extrapolates to unseen inputs."
```

Figure 11: Prompts in DIO-Agent.

H.1 Benchmark Composition

We establish our IO2CodeBench for experiments, informed by [Michaud et al., 2024]. The IO2CodeBench includes Base, Algorithm, Geometry, and Multi-modal levels. Each task corresponds to a deterministic target program defined in a task registry and spans a different transformation pattern over structured inputs. At a high level, the benchmark covers cumulative arithmetic operations, memory-style sequence transductions, parity and bitwise reasoning, multi-stream operations such as base-k addition and binary dot products, and several small symbolic reasoning tasks.

Each task is defined by a ground-truth function f^* together with a task-specific input domain. For the core benchmark, the domain type is one of five categories: integer sequences, bit sequences, paired integer sequences, paired bit sequences, or floating-point sequences. The labels are always generated by executing the oracle,

$$y = f^*(x), \tag{4}$$

so each task is fully specified by executable semantics rather than by natural-language annotation.

H.2 Example Generation

For the core sequence tasks, inputs are sampled automatically according to the task type and range constraints stored in the registry. Sequence lengths are drawn uniformly between 3 and 10. Integer-valued tasks sample each element from a task-specific bounded interval, bit-valued tasks sample each element from $\{0, 1\}$, floating-point tasks sample from a bounded real interval, and two-stream tasks generate two aligned sequences of the same length. The target output is then obtained by directly applying the hidden oracle function to the sampled input.

For the extended tasks, we use task-specific generators. These include bounded integer problems, strings, integer lists, reverse-Polish-expression sequences, bit-vector problems, and simple geometry tasks over points and paths. To improve coverage, the generator first inserts a small set of canonical

edge cases and then fills the remaining examples with random samples from the corresponding domain. This produces a benchmark that is still automatically generated, but less homogeneous than simple iid random sampling.

H.3 Train–Test Split Construction

By default, we construct 8 visible training examples and 15 hidden test examples per task, where each set ensures the 100% test coverage of the task. The training set is generated with random seed 42 and the test set with random seed 999. This separation is hard-coded in the generation pipeline and is also stated explicitly in the prompt shown to the model.

The split procedure is designed to avoid leakage at the input level. For each task, we first sample an oversized candidate pool for training and testing, deduplicate examples by input, and then remove any test input that also appears in the training pool. If the unique pool is still too small, the generator performs an additional top-up with backup seeds. Thus, the benchmark enforces train–test input disjointness whenever the task domain permits it.

Formally, for task τ we construct a visible set $\mathcal{D}_{\text{train}}^{(\tau)}$ and a hidden set $\mathcal{D}_{\text{test}}^{(\tau)}$ such that

$$\mathcal{D}_{\text{train}}^{(\tau)} = \{(x_i, f_\tau^*(x_i))\}_{i=1}^{n_{train}}, \qquad \mathcal{D}_{\text{test}}^{(\tau)} = \{(x_j, f_\tau^*(x_j))\}_{j=1}^{n_{test}}, \tag{5}$$

with disjoint inputs whenever possible. During search, only $\mathcal{D}_{\text{train}}^{(\tau)}$ is visible to the synthesizer; $\mathcal{D}_{\text{test}}^{(\tau)}$ is never revealed to the model.

H.4 Evaluation Protocol

Evaluation is execution-based. Given a synthesized program $\hat{p}$, the evaluator imports the generated Python file, locates the task-specific function, and executes it on every hidden test example. A prediction is marked correct if it exactly matches the expected output. For structured outputs such as lists, comparison is performed recursively element by element; for floating-point values, we allow a small numerical tolerance of 10^{-6}.

For a task with test set $\mathcal{D}_{\text{test}}$, the final task-level accuracy is

$$\text{Acc}_{\text{test}}(\hat{p}) = \frac{1}{|\mathcal{D}_{\text{test}}|} \sum_{(x,y)\in\mathcal{D}_{\text{test}}} \mathbf{1}[\hat{p}(x) = y]. \tag{6}$$

The evaluator also records the number of correct cases, total cases, runtime, and up to three concrete failure examples for diagnosis. In the final hidden evaluation used for benchmark reporting, the selection score equals this held-out accuracy.

At the benchmark level, we report both average hidden-set accuracy and *task success rate*, where a task is counted as solved only if the synthesized program achieves $\text{Acc}_{\text{test}} = 1.0$. This exact-match criterion is important for PBE, since partially correct programs may fit many visible examples while still failing to recover the underlying rule.

H.5 Search-Time vs. Final Evaluation

One final distinction is important. During search, the synthesizer is never optimized directly on the hidden test set. In the baseline setting, search uses a training-only evaluator built from the 8 visible examples. In DIO-Agent, search further decomposes these visible examples into curriculum stages and uses replay examples from earlier stages for regression checking. However, in all cases, the numbers reported in the main tables are obtained from the final hidden evaluator on the 15-example test split. This keeps optimization and reporting cleanly separated and makes the benchmark a true generalization test rather than a memorization exercise.

I Experimental Setup

Benchmark. Our proposed evaluation benchmark comprises 125 IO2Code tasks organized into four difficulty levels designed to test progressively more complex pattern discovery capabilities. The

Base level contains 62 tasks involving scalar arithmetic, list operations, and standard map/filter/reduce patterns. The underlying rule here can typically be expressed in one to three lines of Python. Examples include reversing a list, computing a running sum, or applying element-wise arithmetic. The **Algorithm** level contains 35 tasks requiring algorithmic reasoning, dynamic programming, or multi-step transformations. The mapping from input to output here involves non-obvious algorithmic structure such as sorting by a derived key, computing edit distances, or implementing combinatorial logic. The **Geometry** level contains 18 tasks drawn from geometric computation problems, including area calculations, intersection detection, and coordinate transformations. In addition, the **Multi-modal** level contains 10 tasks where the inputs are images and the outputs correspond to specific numerical properties within the images (such as the number of shapes or spatial relationships). The model is provided with an external multi-modal LLM tool, which it can freely call to extract information from the images for further computational processing [Qwen Team, 2026]. Each task provides 8 visible I/O examples used during search and 15 held-out test cases used exclusively for evaluation. The level structure is designed to separate tasks susceptible to ceiling effects (Base) from those where structured search provides meaningful differentiation (Algorithm, Geometry, Multi-modal). The prompts are shown in Figure 11. In the Autonomous Discovery setting, the model initially constructs 2 I/O examples. Once the evolved code successfully passes all existing examples, the model is permitted to generate additional I/O pairs, increasing the sample count by 2 in each subsequent step. We set the maximum number of iterations to 50 and implement an early stopping strategy if the model consistently satisfies all current examples for five consecutive rounds.

Evaluation Metric. To reflect the differing nature of the tasks, we employ a tiered scoring system. For the Base, Algorithm, and Geometry levels, we report the level Pass Rate [Chen et al., 2021, Austin et al., 2021]. A task is considered as solved only if the generated program $\hat{p}$ correctly satisfies $\hat{p}(x_i) = y_i$ for all 15 held-out test cases. The metric for these levels is the percentage of tasks solved within each category. Conversely, for the Multimodal level, we utilize a more granular Mean Sample Pass Ratio. For each of the 10 multimodal tasks, we calculate the ratio of held-out test cases passed by the model. The final metric for this level is the average of these ratios across all 10 tasks. This approach accounts for the inherent complexity of image-to-value extraction, where partial success across test cases provides a more nuanced measure of model capability. As a secondary metric, we report the Computational Efficiency via the token cost per iteration. This is decomposed into the mean prompt tokens and mean completion tokens across all LLM calls.

Baselines. We compare DIO-Agent against seven baselines spanning the spectrum from classical PBE methods to modern LLM-based evolution-agent baselines. Among these, Direct represents a standard approach where the LLM is prompted to generate code directly based on visible examples [Brown et al., 2020, Chen et al., 2021]. Meanwhile, PBE and E-PBE are methods rooted in the Programming-by-Example paradigm within the software engineering domain [Gulwani, 2016, Verbruggen et al., 2025]. TDD means TDD-guided Generation iteratively. Moreover, FunSearch, AlphaEvolve, and CodeEvolve represent outstanding evolutionary algorithms tailored for the field of code generation [Romera-Paredes et al., 2024, Novikov et al., 2025, Assumpção et al., 2025]. We conduct all experiments here with DeepSeek-v3.2 as the base LLM to ensure fair comparison across methods [Liu et al., 2025].

J Additional Experimental Results

J.1 Variation Across Independent Runs

Table 11 reports a supplemental comparison of Direct, AlphaEvolve, and DIO-Agent over three independent runs. Each entry gives the mean and sample standard deviation across runs. These aggregates are presented separately from the point estimates in the main tables; they do not imply that every baseline, ablation, or supplementary evaluation was repeated three times.

Across the three matched runs, the reported mean improvement of DIO-Agent over AlphaEvolve is 10.51 points in the four-category average, with a 95% confidence interval of $[6.98, 14.04]$. The reported two-sided paired t-test gives $t(2) = 12.81$ and $p = 0.006$. This comparison supports a positive average difference under the evaluated configuration. With only three runs, however, estimates of variability remain limited, and the test does not establish significance separately for each category or for other comparisons in this paper.

Table 11: Supplemental three-run results on IO2CodeBench: mean $\pm$ sample standard deviation. Higher category scores are better; lower token cost is better.

Method	Base	Algorithm	Geometry	Multimodal	Average	Tokens/iter
Direct	36.56 ± 1.86	51.43 ± 4.95	31.48 ± 3.20	12.66 ± 2.31	33.03 ± 1.48	–
AlphaEvolve	53.76 ± 3.72	60.95 ± 1.65	51.85 ± 6.42	23.11 ± 1.92	47.42 ± 0.22	4089.19 ± 71.41
DIO-Agent	61.83 ± 0.93	69.52 ± 3.30	59.26 ± 3.20	41.11 ± 0.77	57.93 ± 1.20	3678.18 ± 104.15

J.2 Evaluation on ProgramBench

We additionally evaluate DIO-Agent on eight repositories from ProgramBench [**?**], extending the evaluation beyond the single-function tasks in IO2CodeBench. Table 12 reports repository-level scores and their unweighted mean. DIO-Agent obtains a mean score of 0.6896, compared with 0.6394 for AlphaEvolve and 0.5212 for the mini-SWE-agent baseline. It exceeds AlphaEvolve on all eight repositories and obtains the highest score on seven; mini-SWE-agent performs best on `fx`. These results provide evidence of transfer to the evaluated repository tasks, while the eight-repository subset does not establish performance across the full benchmark or across repository-scale software engineering more generally.

Table 12: Additional ProgramBench results on eight repositories. Scores are on a 0–1 scale; the best result in each row is bold.

Repository	mini-SWE-agent	AlphaEvolve	DIO-Agent
`zip-password-finder`	0.8409	0.9710	**0.9760**
`zoxide`	0.6118	0.5893	**0.6846**
`srgn`	0.5915	0.6392	**0.7375**
`atlas`	0.2335	0.3926	**0.4375**
`tui-journal`	0.5335	0.6549	**0.6901**
`shellharden`	0.3847	0.8313	**0.8760**
`fx`	**0.5688**	0.4767	0.5023
`walk`	0.4046	0.5598	**0.6124**
Average	0.5212	0.6394	**0.6896**

J.3 Comparison with a General-Purpose Coding Agent

To test whether access to an execution and debugging loop alone accounts for the observed gains, we compare with mini-SWE-agent [**?**] on IO2CodeBench using the same base model and a matched token budget. This baseline can execute, inspect, and debug code. As shown in Table 13, DIO-Agent improves the average score from 44.80 to 58.63, a difference of 13.83 points, with gains in every category. This comparison supports the value of DIO-Agent's specialized search design relative to this coding-agent configuration; it does not isolate the contribution of each individual component.

Table 13: IO2CodeBench comparison with mini-SWE-agent using the same base model and a matched token budget.

Method	Base	Algorithm	Geometry	Multimodal	Average
mini-SWE-agent	46.77	62.86	38.89	30.67	44.80
DIO-Agent	**61.29**	**71.43**	**61.11**	**40.67**	**58.63**

J.4 Sensitivity to the Transformation Ordering

We compare the ordered TPP guidance with removing the guidance and randomly shuffling its ordering while preserving its content. Table 14 shows that the ordered variant exceeds the shuffled variant by approximately 6.5 points on average. Shuffling reduces performance in all four categories despite a similar token cost per iteration. This result indicates that the ordering matters beyond the presence of transformation-related instructions. TPP nevertheless remains soft prompt guidance: the ablation

demonstrates sensitivity to the ordering, not strict compliance at every mutation, exhaustive search over simpler programs, or guaranteed generalization.

Table 14: Effect of TPP ordering. The shuffled variant retains the guidance content while randomizing its order.

Variant	Base	Algorithm	Geometry	Multimodal	Average	Tokens/iter
Ordered TPP (DIO-Agent)	**61.29**	**71.43**	**61.11**	**40.67**	**58.63**	3738.31
Without TPP	**61.29**	68.57	50.00	35.33	53.80	**3369.48**
Shuffled TPP	59.68	68.57	44.44	36.00	52.17	3660.96

J.5 ARC-AGI-3 Pilot and Failure Boundary

We conduct an exploratory evaluation on five ARC-AGI-3 games [**?**], with final evaluation using seed 0 and an 80-action cap. Table 15 reports the official outcomes. Neither Direct nor DIO-Agent completes a level or wins a game, so this pilot provides no evidence of an improvement in official benchmark success. DIO-Agent produces executable policies for all five games, compared with three for Direct, but executability does not establish task solving. The small, single-seed pilot exposes a remaining failure boundary rather than demonstrating broad ARC generalization.

Table 15: Official outcomes in the five-game ARC-AGI-3 pilot. LC: levels completed; WR: win rate. Error means no executable policy was generated. All LC and WR entries are zero.

	Direct				DIO-Agent			
Game	State	LC	WR	Actions	State	LC	WR	Actions
`ls20`	Not finished	0	0	80	Not finished	0	0	80
`ar25`	Error	0	0	0	Not finished	0	0	80
`sp80`	Game over	0	0	30	Game over	0	0	29
`tu93`	Error	0	0	0	Game over	0	0	50
`ft09`	Not finished	0	0	80	Not finished	0	0	80
Executable policies		3/5				5/5		