PolyMind: Exploring Width Scaling for Reflective Reasoning in Language Agents

Heng Zhang[1], Chengyu Zhou[2], Jiajun Wu[3], Xinyuan Liu[4]
Liheng Zhang[1], Yueqi Guo[1], Rui Liu[1], JiaHao Hong[1]
Xuanxun Lian[1], Jinpeng Lu[1], Jin Huang[1]*
[1]South China Normal University [2]LexisNexis
[3]Central South University [4]Northwestern University
2024025450@m.scnu.edu.cn, czhou24190@gmail.com
8210231904@csu.edu.cn, xinyuanliu2024@U.NORTHWESTERN.EDU
huangjin@m.scnu.edu.cn

Abstract

Reflection-based language agents improve reasoning by using feedback to revise previous attempts across multiple trials. Most existing reflection frameworks scale this process by depth, allocating additional test-time compute to longer serial reflection-retry chains. We observe that serial reflection often saturates after rapid early gains. In the default setting, increasing depth from 5 to 10 cycles brings marginal improvement. Isolated reflective workers contain useful revision signals. This raises a central question: how can reflective reasoning benefit from scaling across parallel workers? Parallel reflection introduces two key challenges: overlapping reflective scopes and noisy feedback aggregation. We propose **PolyMind**, a reflective width scaling framework that learns to allocate complementary reflective scopes, execute workers in isolated contexts, and compress parallel feedback into a structured PolyBrief. **PolyMind** trains the planner, workers, and reducer with verifiable outcome feedback and a role-balanced objective that prevents wider rollouts from dominating optimization. Across code generation, mathematical reasoning, and multi-hop question answering, **PolyMind** consistently improves performance under matched budgets, outperforming the strongest depth-reflection baseline by an average of 6.1 percentage points. Further analyses examine scope allocation, feedback compression, and role-balanced learning. The results highlight the importance of jointly allocating reflective computation across depth and width.

1 Introduction

Large language models have shown strong progress on complex reasoning tasks [1, 2, 3]. A key driver behind this progress is test-time scaling [4, 5, 6, 7, 8, 9]. By allocating additional computation during inference, language agents can spend more effort to search, verify, reflect, and revise. Reflection-based reasoning is a representative form of this paradigm. An agent first produces a candidate solution, receives task feedback, writes a reflection, and then attempts the task again [8, 6, 7]. This simple loop has been effective across programming [8], mathematical reasoning [10], decision-making [11], and multi-hop question answering [12, 13].

Most reflection-based agents spend additional compute in a serial manner. A longer budget is usually used for more reflection-retry cycles along the same chain [6, 8, 7, 14]. This depth-oriented allocation is intuitive, since each new attempt can use feedback from previous attempts. However, recent studies also show that self-reflection can become repetitive or inaccurate after several rounds [15, 14, 16, 17].

*Corresponding author.

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

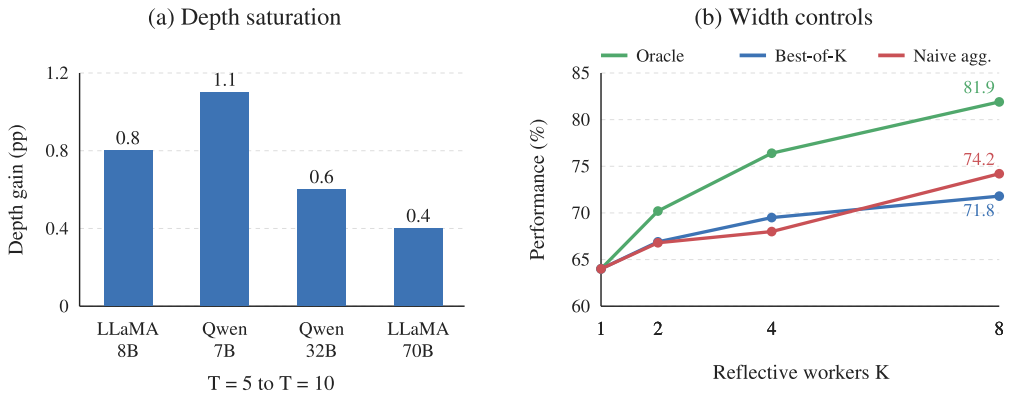


Figure 1: Depth saturation and reflective width scaling. (a) The gain from 5 to 10 serial cycles is at most 1.1 percentage points across four backbones. (b) On the five-dataset preliminary setting with LLaMA-3.1-8B, Oracle Width evaluates K revisions, each conditioned on a different worker reflection, and succeeds if any revision is correct. Best-of-K samples the actor without reflection, while naive aggregation combines all worker outputs for one revision. The width curves use one reflection cycle with varying K; the depth diagnostic varies serial cycles with one worker. These are independent scaling diagnostics. Appendix B compares methods under matched end-to-end budgets.

This pattern leaves an important scaling question open. How should reflective compute be allocated across serial cycles and parallel workers?

We begin with a study of this question. Figure 1 reveals three observations. First, serial depth quickly saturates. The gain from 5 to 10 serial cycles is 0.8 percentage points in the default setting and at most 1.1 points across four tested backbones. Second, oracle width keeps improving as the number of isolated reflective workers increases. Third, naive aggregation at $K = 8$ leaves a 7.7-point gap from Oracle Width. Oracle Width uses pass@K over K reflection-conditioned actor revisions, Best-of-K samples the actor K times without reflection, and naive aggregation combines worker feedback for one actor revision (Appendix C). These observations motivate allocating complementary scopes and compressing feedback from parallel workers.

How can parallel reflective work be allocated and integrated to improve reasoning?

The challenge is to integrate parallel reflective feedback into a concise context for actor revision. Direct concatenation makes the revision context long and noisy. Generic summarization can remove minority feedback that is useful for revision. A width-scalable reflection system must therefore decide what each worker should examine, how much budget each cycle should spend, and how parallel feedback should be compressed into a concise signal for revision.

We propose `PolyMind`, a framework for exploring width scaling in reflective reasoning. At each reflection cycle, a learned planner generates a compact reflective width decision. A scope matrix assigns complementary reflective scopes to isolated workers. A width budget controls the number of workers and the length of their outputs. A compression mode reduces all worker outputs into a structured PolyBrief with Keep, Revise, and Verify fields. The actor then revises the solution with this brief and receives new feedback. The planner, workers, and reducer are trained with verifiable outcome feedback, so the system learns width decisions that improve final task success. This design allocates reflective compute across depth and width.

Contribution. Our contributions are summarized as follows:

❶ **Scaling Discovery.** We formulate reflective width scaling as a test-time compute allocation problem for language agents. Controlled studies across five datasets show that depth-only reflection saturates quickly and that reflective compute can also be spent effectively across parallel workers.

❷ **Reflective Width Execution.** We propose `PolyMind`, a reflective reasoning framework that represents each cycle as a compact width decision, assigns complementary scopes to isolated workers, and compresses parallel feedback into a structured PolyBrief for revision.

❸ **Role-Balanced Learning.** We train the planner, workers, and reducer with verifiable outcome feedback under a role-balanced objective. This objective prevents rollouts with more workers from dominating optimization and encourages useful width allocation.

❹ **Empirical Evidence.** Experiments across programming, mathematical reasoning, and multi-hop question answering show that `PolyMind` improves the average score by 6.1 percentage points over the strongest depth-reflection baseline under matched budgets. Further experiments show width gains across four backbones.

2 Related Work

LLM Reasoning and Test-Time Scaling. Large language models have demonstrated strong multi-step reasoning ability when guided by appropriate prompting and inference-time computation. Chain-of-Thought prompting elicits explicit intermediate reasoning steps and improves performance on complex tasks [4]. Self-Consistency further samples multiple reasoning paths and aggregates them by majority voting, showing that spending more inference compute can improve robustness [5]. ReAct interleaves reasoning with external actions, enabling language agents to reason and interact with tools or environments in a unified loop [11]. Other methods extend inference-time reasoning through structured search, such as Tree of Thoughts and Graph of Thoughts [7, 18], or through iterative self-feedback and reflection [6, 8]. More recently, test-time compute scaling has become a central direction for improving reasoning by allocating more computation during inference [19, 20, 21]. `PolyMind` investigates reflective width as a complementary dimension of test-time compute allocation.

Reflection-Based Language Agents. Reflection-based agents improve task performance by using feedback from previous attempts to guide subsequent revisions. Self-Refine and Reflexion introduce iterative feedback and memory mechanisms, allowing agents to refine their outputs across multiple trials [6, 8]. Retroformer learns a retrospective model with policy optimization to improve feedback-driven refinement [14]. Self-RAG trains reflection tokens for generation and retrieval decisions [22], and ExpeL extracts reusable experience from successful trajectories for cross-task transfer [23]. Recent studies also show that self-reflection can become repetitive or inaccurate, motivating methods that enrich reflection through diverse prompts, cross-sample memory, or parametric reflective memory [15, 16, 17]. Existing reflection methods advance feedback generation, storage, and retrieval. `PolyMind` studies the allocation and compression of reflective compute at test time. It treats each reflection cycle as a width decision that assigns complementary reflective scopes to isolated workers and compresses their outputs into a concise revision brief.

Parallel and Width Scaling in Agentic Systems. Multi-agent systems provide a natural way to distribute reasoning and tool-use computation across multiple roles or workers [24, 25]. Prior systems often rely on hand-crafted workflows, role assignments, or turn-taking interactions to coordinate agents, and have shown strong performance in collaborative reasoning, software engineering, and information-seeking tasks [26, 27, 28]. Recent work on broad information seeking and wide search further highlights that some agentic tasks benefit from horizontal execution, where multiple workers handle partially independent subtasks before their outputs are aggregated [29, 30, 31]. These works motivate width as an important scaling dimension for agentic computation. `PolyMind` brings this perspective to reflective reasoning. It applies width scaling to reflective computation through isolated workers with complementary scopes and learns to compress their feedback for revision.

3 Preliminaries

3.1 Reflection Based Reasoning

We consider a language agent built on a pretrained language model. For a task input x, the actor p_θ generates an answer, receives feedback from a task evaluator E, and uses natural language reflections to guide later attempts. At reflection cycle t, the agent has accumulated previous reflections $r_{1:t-1}$ and generates a new answer y_t by conditioning on the task and this reflection history.

$$y_t \sim p_\theta(\cdot \mid x, r_{1:t-1}) \tag{1}$$

The evaluator returns feedback $e_t = E(x, y_t)$. This feedback can be unit test results in code tasks, answer correctness in math tasks, or verification feedback in question answering tasks. Existing

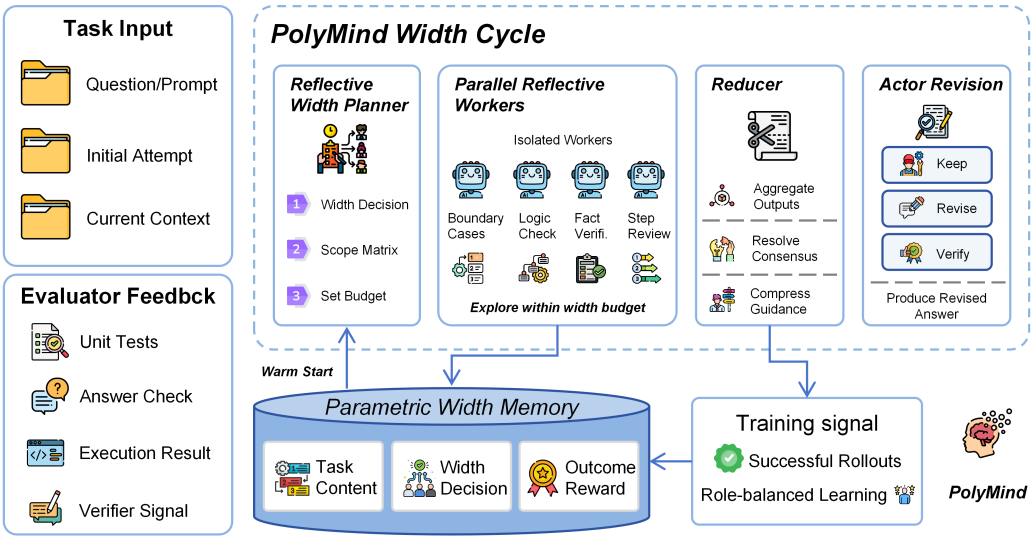


Figure 2: Overview of **PolyMind**.

reflection based agents usually spend more test time compute by extending this serial chain. The number of cycles increases, and each new answer is generated after reading the previous reflection history. We refer to this common allocation pattern as depth scaling for reflective reasoning.

3.2 Depth and Width in Reflective Compute

Depth scaling treats reflection as a single growing chain. Our goal is to study a broader allocation of reflection compute. A cycle may use multiple isolated reflective workers, each with a local token budget, before their outputs are compressed for revision. Let T be the number of serial reflection cycles, K_t be the number of workers used at cycle t, and L_t be the generation cap assigned to each worker. The worker-generation budget can be written as

$$\mathcal{C}_{\text{work}} = \sum_{t=1}^{T} K_t L_t \tag{2}$$

The serial reflection setting corresponds to $K_t = 1$ for every cycle. Reflective width scaling allocates this budget across T and K_t. End-to-end token usage also includes the actor, planner, and reducer.

4 Methodology

4.1 Overview

The core idea of **PolyMind** is learned scope allocation and feedback compression for parallel reflection. Naive width can produce multiple reflections, but those reflections often repeat similar concerns or become difficult to use after aggregation. **PolyMind** addresses this by learning how to allocate reflective scopes and how to compress parallel feedback into a concise revision signal. The actor p_θ remains responsible for producing task answers. A planner p_ϕ produces a reflective width decision. A worker model p_ψ executes each assigned reflective scope in an isolated context. A reducer p_ρ compresses worker outputs into a PolyBrief for revision. At cycle t, the current context consists of the original task x, the previous answer y_{t-1}, and the evaluator feedback e_{t-1}.

$$u_t = (x, y_{t-1}, e_{t-1}) \tag{3}$$

For the first cycle, y_0 is the initial actor output and $e_0 = E(x, y_0)$. If e_0 already indicates success, the agent returns y_0. Otherwise, **PolyMind** starts a reflective width cycle based on u_t.

This decomposition of reflective width into planning, execution, and compression follows the main observation from the pilot study. Naive width often repeats reflective scopes and produces noisy revision contexts. The system must first direct different workers toward different aspects of the current

attempt. It must also distill multiple outputs into a form that the actor can reliably use. The planner provides the first capability by producing a scope matrix before execution. The reducer provides the second by compressing parallel outputs into a fixed structure brief. Training these components with outcome feedback closes the loop and allows the system to improve its width allocation over time.

4.2 Reflective Width Decision

A reflective width cycle begins by deciding what should be examined in parallel and how the resulting feedback should be compressed. The planner samples a decision Ω_t from the current context u_t.

$$\Omega_t \sim p_\phi(\cdot \mid u_t) \tag{4}$$

The decision contains a scope matrix $\mathcal{S}_t$, a width budget $\mathcal{B}_t$, and a compression mode $\mathcal{C}_t$.

$$\Omega_t = (\mathcal{S}_t, \mathcal{B}_t, \mathcal{C}_t) \tag{5}$$

The scope matrix is the carrier of reflective width. It contains K_t rows, and each row assigns one worker a scope $s_{t,i}$ and an output constraint $c_{t,i}$. The scope tells the worker what to examine, and the constraint tells it what kind of reflective output is expected.

$$\mathcal{S}_t = [(s_{t,1}, c_{t,1}), \ldots, (s_{t,K_t}, c_{t,K_t})] \in \mathcal{V}^{K_t \times 2} \tag{6}$$

The text space is denoted by $\mathcal{V}$. The budget $\mathcal{B}_t = (K_t, L_t)$ ties the number of workers to the matrix size and assigns each worker at most L_t tokens. The compression mode $\mathcal{C}_t$ determines how worker outputs are reduced. In the main method, the reducer uses a structured mode and produces a PolyBrief with Keep, Revise, and Verify fields.

The content of the scope matrix adapts to the task domain. For programming tasks, common scopes include boundary condition handling, loop behavior, return type consistency, and input format assumptions. Each scope directs a worker to examine one aspect of the current solution. For mathematical reasoning, scopes may separate condition use from computation, or isolate a specific derivation step for independent verification. For multi-hop question answering, scopes may assign workers to verify different factual claims, relation links, or answer normalization requirements. In all cases, the planner generates task-specific reflective scopes conditioned on the current task context and evaluator feedback e_{t-1}.

Reflective width is represented as a structured decision over what to examine, how many workers to use, and how their outputs should be compressed.

4.3 Parallel Reflection and Compression

After the decision is generated, each row of $\mathcal{S}_t$ is mapped to one isolated worker. Worker i receives the current context u_t, its scope $s_{t,i}$, its constraint $c_{t,i}$, and the token budget L_t. It generates a local reflection $z_{t,i}$ without seeing the outputs of other workers.

$$z_{t,i} \sim p_\psi(\cdot \mid u_t, s_{t,i}, c_{t,i}, L_t) \tag{7}$$

Isolation makes the workers depend on the planner decision rather than on one another. Once all K_t workers finish, the reducer receives their outputs $z_{t,1:K_t}$ and the compression mode $\mathcal{C}_t$. It produces a PolyBrief b_t that preserves useful feedback in a short form.

$$b_t \sim p_\rho(\cdot \mid u_t, z_{t,1:K_t}, \mathcal{C}_t) \tag{8}$$

The PolyBrief is intentionally shorter than the combined worker outputs. A typical cycle with several workers may produce hundreds of tokens of parallel feedback, but the brief condenses them into a compact structure. The Keep field anchors correct parts of the current answer, so the actor does not unnecessarily change components that already work. The Revise field presents concrete repair directions supported by worker outputs. When workers disagree, the reducer favors claims supported by multiple outputs and avoids exposing the actor to unresolved conflict. The Verify field states conditions that the revised answer should satisfy, giving the actor a lightweight checklist for the next attempt. This structure is designed to be actionable. It tells the actor what is safe to keep, what should change, and how to check the revised answer.

The actor then revises the task answer using the original task and this brief.

$$y_t \sim p_\theta(\cdot \mid x, b_t) \tag{9}$$

The evaluator returns new feedback $e_t = E(x, y_t)$. If the task is solved, the process stops. Otherwise, the next cycle builds a new context from x, y_t, and e_t, allowing `PolyMind` to allocate width again under the remaining budget. The complete inference procedure is summarized in Algorithm 1.

Algorithm 1 `PolyMind` Reflective Width Execution

Require: task x, actor p_θ, evaluator E, planner p_ϕ, worker model p_ψ, reducer p_ρ, maximum cycles $T_{\max}$

$y_0 \sim p_\theta(\cdot \mid x)$
$e_0 \leftarrow E(x, y_0)$
for $t = 1$ to $T_{\max}$ **do**
  if $e_{t-1} = \text{pass}$ **then**
    return y_{t-1}
  end if
  $u_t \leftarrow (x, y_{t-1}, e_{t-1})$
  $\Omega_t = (\mathcal{S}_t, \mathcal{B}_t, \mathcal{C}_t) \sim p_\phi(\cdot \mid u_t)$
  Parse $\mathcal{B}_t = (K_t, L_t)$
  Parse $\mathcal{S}_t$ into $\{(s_{t,i}, c_{t,i})\}_{i=1}^{K_t}$
  for $i = 1$ to K_t **in parallel do**
    $z_{t,i} \sim p_\psi(\cdot \mid u_t, s_{t,i}, c_{t,i}, L_t)$
  end for
  $b_t \sim p_\rho(\cdot \mid u_t, z_{t,1:K_t}, \mathcal{C}_t)$
  $y_t \sim p_\theta(\cdot \mid x, b_t)$
  $e_t \leftarrow E(x, y_t)$
end for
return $y_{T_{\max}}$

4.4 Learning Reflective Width

The planner, workers, and reducer are trained with verifiable outcome feedback. We freeze the actor in the main setting. For a fixed context u_t, we sample G width rollouts. The g th rollout contains the sampled decision Ω_t^g, the worker outputs $z_{t,1:K_g}^g$, the brief b_t^g, and the revised answer y_t^g.

$$\tau_g = (\Omega_t^g, z_{t,1:K_g}^g, b_t^g, y_t^g) \tag{10}$$

The evaluator provides a binary outcome reward R_g for the revised answer. Rewards are normalized inside the group to obtain the rollout advantage $\hat{A}_g$.

$$R_g = \mathbb{I}[E(x, y_t^g) = \text{pass}] \qquad \hat{A}_g = \frac{R_g - \mu_R}{\sigma_R + \epsilon} \tag{11}$$

The quantities μ_R and σ_R are the mean and standard deviation of the G rewards, and ϵ is a small constant. Let $\mathcal{J}(o, \hat{A}_g)$ denote the token averaged clipped policy objective for a generated sequence o under advantage $\hat{A}_g$. The planner output is o_g^{plan}, worker i produces $o_{g,i}^{\text{work}}$, and the reducer output is o_g^{red}. We balance the three roles by averaging over the planner, all workers, and the reducer.

$$\mathcal{L}_g = \frac{1}{3}\left(\mathcal{J}(o_g^{\text{plan}}, \hat{A}_g) + \frac{1}{K_g}\sum_{i=1}^{K_g}\mathcal{J}(o_{g,i}^{\text{work}}, \hat{A}_g) + \mathcal{J}(o_g^{\text{red}}, \hat{A}_g)\right) \tag{12}$$

Averaging the worker term by K_g prevents rollouts with more workers from dominating the gradient because they produce more tokens. The final objective averages $\mathcal{L}_g$ over the sampled group.

$$\mathcal{L} = \frac{1}{G}\sum_{g=1}^{G}\mathcal{L}_g \tag{13}$$

The online objective trains the system to make useful width decisions for the sampled tasks. Effective decomposition patterns can also recur across related problems. A scope structure that works well for a boundary handling issue in one programming task may help with a similar input condition in another task. A decomposition that separates factual claims in one multi-hop question may also help in another question with similar reasoning structure. To capture such cross sample regularity, we maintain a compact record set for parametric width learning. Each record stores a task context u_j, the sampled decision Ω_j, and the outcome reward R_j.

$$\mathcal{D}_{\text{width}} = \{(u_j, \Omega_j, R_j)\}_{j=1}^{N} \tag{14}$$

After training epochs, high reward records are used to fine tune a lightweight module F_η that maps task contexts to effective width decisions. At inference time, the planner can use $F_\eta(u_t)$ as a prior

before generating its own decision. This parametric component provides a warm start based on cross sample allocation patterns, while the planner can still deviate from the prior when the current task requires a different decomposition. In this way, `PolyMind` combines the adaptivity of outcome feedback with the reuse of learned width patterns.

5 Experiments

5.1 Experiment Setup

Datasets and Metrics. We evaluate `PolyMind` on three categories of tasks that provide verifiable feedback and require different reflective workloads. The **code generation** category uses HumanEval [32] and MBPP [33], where execution results provide direct feedback and pass@1 is reported. The **mathematical reasoning** category uses GSM8K [34] and MATH [35], where final answer checking provides binary correctness and accuracy is reported. The **multi-hop question answering** category uses HotpotQA [12] and 2WikiMultiHopQA [13], where answers require multiple factual checks and performance is measured by exact match. These tasks evaluate reflective compute allocation across serial cycles and parallel workers.

Baselines. We compare `PolyMind` with four groups of baselines. ❶ **Direct reasoning** baselines include Vanilla, CoT [4], and Self-Consistency [5]. ❷ **Depth-reflection** baselines include Self-Refine [6], Reflexion [8], and Retroformer [14], which spend extra compute on serial reflection and retry cycles. ❸ **Naive-width** baselines run several reflective workers in parallel and combine their outputs by direct union, generic summarization, or voting. ❹ **Reflection-enhanced** baselines include DoT [16] and ParamMem [36], which enrich reflective signals through diversity or memory. Oracle Width serves as a pass@K upper bound in the analysis (Appendix C).

Implementation Details. Unless otherwise specified, all methods use LLaMA-3.1-8B as the backbone and the same answer extraction scripts. The actor is frozen and has no adapter. The planner, workers, and reducer share the backbone with three role-specific rank 16 LoRA adapters, trained with learning rate 3×10^{-5} and group size $G = 8$. The default setting uses maximum cycles $T_{\max} = 2$, reflective width $K = 4$, and worker generation cap $L = 256$. End-to-end token usage includes all four roles and totals 4,314 for `PolyMind` and 4,316 for Retroformer. Exact training settings, end-to-end accounting, and capacity controls are given in Appendices D and B. We repeat each experiment with three seeds and report averaged performance.

5.2 Main Result (RQ1)

RQ1: Can learned reflective width improve reasoning under the same reflection budget?

Table 1 shows that learned reflective width improves performance across all three task categories. The improvement also appears on mathematical reasoning and multi-hop question answering, where feedback is more compact and the benefit depends on how the system uses it. Compared with the strongest depth-reflection baseline, `PolyMind` improves the average score from 66.5 to 72.6 under a similar token budget. The average score also exceeds those of the naive-width variants. `PolyMind` learns what workers should examine and how their feedback should be compressed.

Finding 1. Learned reflective width consistently outperforms depth-only and naive-width reflection under matched budgets, suggesting that reflective reasoning benefits from broader compute allocation when the width is explicitly learned.

5.3 Model Analysis (RQ2)

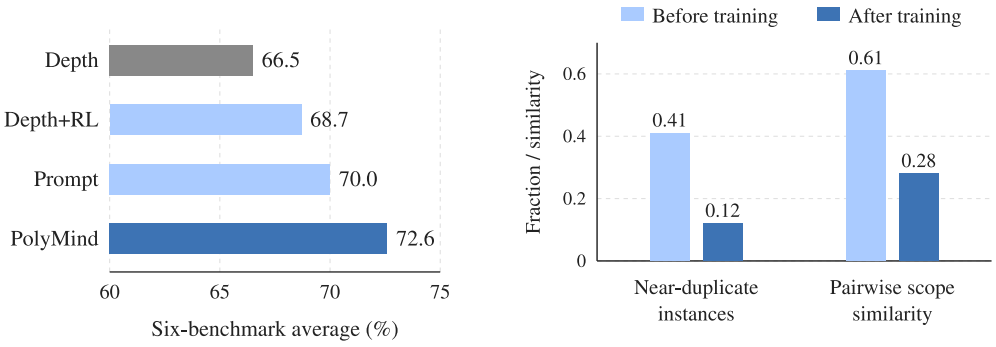
RQ2: Does `PolyMind` turn useful width signals into deployable gains?

Figure 3a compares trained and untrained depth and width variants. With the same frozen actor and matched total tokens, a depth baseline trained with a 63M-parameter LoRA budget reaches 68.7, while untrained `PolyMind`-Prompt reaches 70.0. Training `PolyMind` adds a further 2.6 points, reaching 72.6. Appendix C gives the sampling diagnostic.

Figure 3b further examines the allocation behavior. Before training, the planner produces near-duplicate scopes on 41% of instances with mean pairwise similarity 0.61. After training, these values

Table 1: Main results across code generation, mathematical reasoning, and multi-hop question answering under the worker-budget protocol. The best result in each column is shaded. Token cost is the relative prompt-token budget compared with vanilla generation. Appendix B gives end-to-end token usage for each module.

Type	Method	HumanEval	MBPP	GSM8K	MATH	HotpotQA	2Wiki	Avg.	Tokens
Direct	Vanilla	49.8	55.7	68.4	37.5	52.1	48.9	52.1	1.0×
	CoT	55.6	61.8	74.9	43.2	58.4	54.6	58.1	1.2×
	Self-Consistency	61.3	66.9	79.8	47.1	61.0	57.8	62.3	5.0×
Depth	Self-Refine	64.0	68.7	80.9	48.1	63.2	60.4	64.2	4.1×
	Reflexion	66.1	70.2	82.1	49.5	64.9	62.0	65.8	4.4×
	Retroformer	67.0	71.1	82.6	50.3	65.5	62.7	66.5	4.5×
Enhanced Reflection	DoT	67.8	72.0	83.1	51.0	66.2	63.5	67.3	4.5×
	ParamMem	69.1	73.4	84.0	52.2	67.4	64.2	68.4	4.6×
Naive Width	Parallel-Union	67.4	71.8	83.5	51.8	66.5	63.8	67.5	4.7×
	Parallel-Summary	68.2	72.1	83.8	52.4	67.0	64.4	68.0	4.6×
	Parallel-Vote	68.5	72.8	84.1	52.0	66.8	64.1	68.1	4.6×
Learned Width	`PolyMind`-Prompt	70.5	74.7	85.5	54.2	68.9	66.1	70.0	4.7×
	`PolyMind`-SFT	72.2	76.1	86.3	55.8	70.2	67.2	71.3	4.8×
	`PolyMind`	**73.8**	**77.6**	**87.1**	**57.2**	**71.5**	**68.3**	**72.6**	4.8×



(a) Capacity-controlled comparison

(b) Scope allocation before and after training

Figure 3: Model analysis of reflective width. (a) The same frozen actor is used throughout. Depth+RL and trained `PolyMind` each have 63M trainable parameters, while the two untrained variants have none; total tokens are matched at approximately 4,300. (b) Planner training reduces near-duplicate scopes and mean pairwise scope similarity.

fall to 12% and 0.28. Replacing PolyBrief with generic compression reduces the six-benchmark average from 72.6 to 68.9 in Table 2. Appendix G discusses cases where the reducer still loses useful feedback.

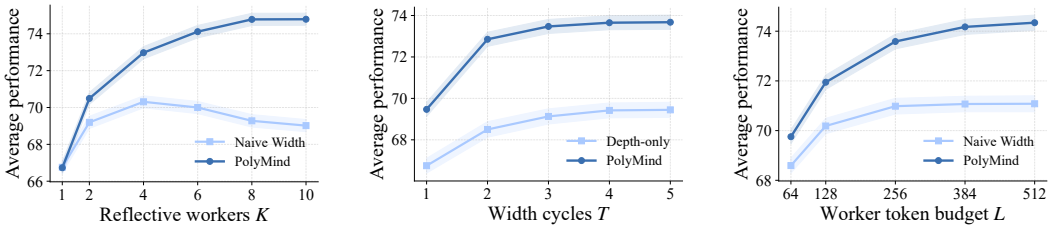
Finding 2. With the same 63M trainable-parameter budget, `PolyMind` reaches an average score of 72.6 versus 68.7 for Depth+RL.

5.4 Hyperparameter Analysis (RQ3)

RQ3: How should a fixed reflection budget be allocated across depth, width, and worker budget?

Figure 4 studies how reflective compute should be allocated. Increasing K gives clear gains until the width becomes large enough that the reducer must handle more repeated or weak signals. Increasing the number of cycles T helps at first but saturates quickly. Increasing the worker token budget L improves performance when workers are too short to produce actionable feedback, but the benefit becomes small after $L = 256$. These results suggest that PolyMind benefits more from distributing compute across complementary scopes than from excessively lengthening each individual reflection.

Finding 3. The best budget allocation in these experiments is moderately wide and shallow.



(a) Reflective width K (b) Width cycles T (c) Worker budget L

Figure 4: Hyperparameter analysis under controlled reflection budgets. Increasing width gives the largest gain before saturation, a small number of width cycles is sufficient, and excessively long worker outputs provide limited additional benefit.

Table 2: Ablation study showing three representative tasks and the average over all six benchmarks. The largest drops come from removing scope allocation and structured PolyBrief.

Variant	HumanEval	MATH	HotpotQA	All-six avg.
`PolyMind`	**73.8**	**57.2**	**71.5**	**72.6**
w/o Scope Matrix	69.1	52.0	66.7	68.4
w/o Isolation	71.0	54.6	69.1	70.1
w/o Structured PolyBrief	69.5	52.8	67.5	68.9
w/o Verify Field	71.7	55.5	70.0	71.0
w/o Role-balanced Loss	71.4	55.0	69.6	70.7
w/o Parametric Width Memory	72.4	56.0	70.4	71.2
fixed K, L	72.0	55.7	70.1	70.9

5.5 Ablation Study (RQ4)

RQ4: Which components are necessary for turning reflective width into reliable gains?

Table 2 shows that each component contributes to turning width into stable gains. Removing the scope matrix causes the largest drop, since workers no longer receive explicit assignments and tend to repeat similar checks. Replacing PolyBrief with generic compression also causes a large drop. Removing isolation weakens the benefit of width because workers become less independent. Removing role-balanced learning hurts optimization, since rollouts with more workers can dominate the gradient. Removing parametric width memory causes a smaller drop.

Finding 4. Reflective width performance depends on how parallel reflections are assigned, isolated, compressed, and learned from feedback.

6 Conclusion

We investigate reflective width scaling for language agents. Our analysis identifies depth saturation and useful revision signals among isolated workers. `PolyMind` learns to allocate reflective scopes, execute workers in isolation, and compress feedback into a structured PolyBrief. Across code generation, mathematical reasoning, and multi-hop question answering, learned reflective width improves performance under matched budgets.

Acknowledgements and Disclosures

This work was supported by the Natural Science Foundation of Guangdong Province, China (Grant No. 2026A1515010176).

The authors declare no competing interests.

References

[1] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. Advances in neural information processing systems, 33:1877–1901, 2020.

[2] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. Journal of Machine Learning Research, 24(240):1–113, 2023.

[3] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971, 2023.

[4] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. Advances in neural information processing systems, 35:24824–24837, 2022.

[5] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. arXiv preprint arXiv:2203.11171, 2022.

[6] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. Advances in Neural Information Processing Systems, 36:46534–46594, 2023.

[7] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. Advances in neural information processing systems, 36:11809–11822, 2023.

[8] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. Advances in Neural Information Processing Systems, 36:8634–8652, 2023.

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

[10] Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In The Twelfth International Conference on Learning Representations, 2023.

[11] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In International Conference on Learning Representations (ICLR), 2023.

[12] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W Cohen, Ruslan Salakhutdinov, and Christopher D Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. arXiv preprint arXiv:1809.09600, 2018.

[13] Xanh Ho, Anh-Khoa Duong Nguyen, Saku Sugawara, and Akiko Aizawa. Constructing a multi-hop qa dataset for comprehensive evaluation of reasoning steps. arXiv preprint arXiv:2011.01060, 2020.

[14] Weiran Yao, Shelby Heinecke, Juan Carlos Niebles, Zhiwei Liu, Yihao Feng, Le Xue, Rithesh Murthy, Zeyuan Chen, Jianguo Zhang, Devansh Arpit, et al. Retroformer: Retrospective large language agents with policy gradient optimization. arXiv preprint arXiv:2308.02151, 2023.

[15] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. arXiv preprint arXiv:2310.01798, 2023.

[16] Vijay Lingam, Behrooz Omidvar-Tehrani, Sujay Sanghavi, Gaurav Gupta, Sayan Ghosh, Linbo Liu, Luke Huan, and Anoop Deoras. Enhancing language model agents using diversity of thoughts. 2025.

[17] Onat Ozer, Grace Wu, Yuchen Wang, Daniel Dosti, Honghao Zhang, and Vivi De La Rue. Mar: Multi-agent reflexion improves reasoning abilities in llms. arXiv preprint arXiv:2512.20845, 2025.

[18] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. In Proceedings of the AAAI conference on artificial intelligence, volume 38, pages 17682–17690, 2024.

[19] Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling parameters for reasoning. In The Thirteenth International Conference on Learning Representations, 2025.

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

[21] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. Nature, 645(8081):633–638, September 2025.

[22] Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-rag: Learning to retrieve, generate, and critique through self-reflection. 2024.

[23] Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 38, pages 19632–19642, 2024.

[24] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. arXiv preprint arXiv:2308.08155, 3(4), 2023.

[25] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. International Conference on Learning Representations, ICLR, 2024.

[26] Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative agents for "mind" exploration of large language model society. In Advances in Neural Information Processing Systems, volume 36, pages 51991–52008, 2023.

[27] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 15174–15186, Bangkok, Thailand, August 2024. Association for Computational Linguistics.

[28] Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and

Saleema Amershi. Magentic-one: A generalist multi-agent system for solving complex tasks, 2024.

[29] Ryan Wong, Jiawei Wang, Junjie Zhao, Li Chen, Yan Gao, Long Zhang, Xuan Zhou, Zuo Wang, Kai Xiang, Ge Zhang, Wenhao Huang, Yang Wang, and Ke Wang. WideSearch: Benchmarking agentic broad info-seeking, 2025.

[30] Zelai Xu, Zhexuan Xu, Ruize Zhang, Chunyang Zhu, Shi Yu, Weilin Liu, Quanlu Zhang, Wenbo Ding, Chao Yu, and Yu Wang. WideSeek-R1: Exploring width scaling for broad information seeking via multi-agent reinforcement learning, 2026.

[31] Mingju Chen, Guibin Zhang, Heng Chang, Yuchen Guo, and Shiji Zhou. A-MapReduce: Executing wide search via agentic mapreduce, 2026.

[32] 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. arXiv preprint arXiv:2107.03374, 2021.

[33] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. arXiv preprint arXiv:2108.07732, 2021.

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

[35] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. arXiv preprint arXiv:2103.03874, 2021.

[36] Tianjun Yao, Yongqiang Chen, Yujia Zheng, Pan Li, Zhiqiang Shen, and Kun Zhang. Parammem: Augmenting language agents with parametric reflective memory, 2026.

[37] 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. NeurIPS, 2021.

[38] Qwen Team. Qwen2.5: A party of foundation models. https://qwenlm.github.io/blog/qwen2.5/, September 2024. Official citation supplied by the Qwen2.5-7B-Instruct model card.

[39] Qwen Team. Qwen2.5-7B-Instruct: Model card. https://huggingface.co/Qwen/Qwen2.5-7B-Instruct. Accessed November 3, 2026.

[40] Qwen Team. Qwen2.5-32B-Instruct: Model card. https://huggingface.co/Qwen/Qwen2.5-32B-Instruct. Accessed November 3, 2026.

[41] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. arXiv e-prints, pages arXiv–2407, 2024.

A Limitations, Statistical Significance, and Compute Resources

Limitations. `PolyMind` relies on the existence of useful reflective signals among isolated workers. When a task failure cannot be corrected through feedback-based revision, or when all workers repeatedly produce similar reflections, increasing width provides limited benefit. The method also introduces additional inference cost because each reflective cycle may invoke multiple workers before revision. The wall-clock latency of parallel execution depends on the serving infrastructure. Our current evaluation focuses on code generation, mathematical reasoning, and multi-hop question answering, where verifiable feedback is available. Extending `PolyMind` to open-ended generation, multimodal reasoning, and long-horizon tool-interactive agents remains an important direction for future work.

Statistical significance. For the main experimental results, we average performance over three runs with different random seeds.

Compute resources. All experiments are conducted on internal GPU servers. The main training experiments use NVIDIA A100 GPUs with 80GB memory. For the default setting, each training run uses 8 A100 GPUs, including initial actor generation, planner sampling, parallel worker reflection, reducer compression, evaluator feedback, and LoRA-based parameter updates. The major additional cost of `PolyMind` comes from reflective width execution, since each cycle uses K isolated workers with a per-worker token cap L. The worker-generation budget scales linearly with the number of cycles T, the number of workers K, and the worker generation cap L. Appendix B gives end-to-end token usage across all four roles, FLOPs, and wall-clock latency.

Broader impacts. `PolyMind` aims to improve the efficiency and reliability of reflection-based language agents by allocating test-time compute more effectively. This can reduce the cost of deploying reasoning agents under a fixed performance target and may make reflective reasoning more accessible for smaller models. At the same time, stronger reasoning agents may also increase the capability of systems used in unsafe or malicious applications if deployed without safeguards. We therefore recommend using `PolyMind` together with standard safety evaluation, task-specific access control, controlled release practices, and downstream monitoring.

B End-to-End Compute and Capacity Controls

The worker-generation budget in Equation 2 allocates reflection across depth and width, with each worker generating at most L tokens. End-to-end token usage sums the costs of the actor, planner, workers, and reducer. Table 3 gives token usage for each module, FLOPs, and latency averaged over six benchmarks on LLaMA-3.1-8B.

Table 3: End-to-end token usage, FLOPs, and latency on LLaMA-3.1-8B. FLOPs are relative to vanilla generation. The Retroformer, Parallel-Vote, and `PolyMind` token totals are matched within 0.1%.

Method	Actor	Planner	Worker	Reducer	Total	FLOPs	Latency	Avg.
Vanilla	512	–	–	–	512	1.00×	4.1 s	52.1
Self-Refine, $T = 4$	1,280	–	2,640	–	3,920	7.81×	26.9 s	64.2
Retroformer, $T = 5$	1,536	–	2,780	–	4,316	8.59×	31.6 s	66.5
ParamMem	1,280	–	2,910	180	4,370	8.71×	22.4 s	68.4
Parallel-Vote, $K = 4$	1,024	–	3,120	168	4,312	8.58×	14.2 s	68.1
`PolyMind`, $K = 4, T = 2$	1,024	214	2,880	196	4,314	8.61×	13.8 s	72.6

The planner and reducer together account for 410 tokens, about 9.5% of the `PolyMind` total, within the matched end-to-end budget. Retroformer and `PolyMind` use 4,316 and 4,314 tokens, respectively, and their FLOPs differ by less than 2%. Their latencies are 31.6 and 13.8 seconds. Parallel workers shorten the critical path, while serial reflection must finish each cycle before starting the next.

Giving the depth baseline 30% more total tokens, 5,610 at $T = 7$, raises its average to 66.9.

Capacity-matched attribution. The planner, worker, and reducer share one backbone and use three role-specific rank-16 LoRA adapters, with 63M trainable parameters in total. The depth control uses one rank-48 adapter with the same 63M parameter budget, trained with the same reward and data. Every row in Table 4 uses the same frozen actor without an adapter.

Untrained `PolyMind` exceeds the trained depth control by 1.3 points, and role training adds 2.6 points to `PolyMind`.

C Oracle Width and Repeated Actor Sampling

Oracle Width evaluates K actor revisions, each conditioned on a different worker reflection, and counts an instance as solved if any revision is correct. It is a pass@K upper bound. No-reflection Best-of-K also uses K actor samples, but those samples receive no worker reflection. Naive aggregation

Table 4: Capacity-controlled comparison on the six-benchmark setting. Untrained variants have no role-training parameters. The two trained variants have the same trainable-parameter budget, and all rows use the same frozen actor.

Variant	Trainable parameters	Total tokens	Avg.
Retroformer, untrained	0	4,316	66.5
Depth + RL training	63M	4,316	68.7
`PolyMind`-Prompt, untrained	0	4,314	70.0
`PolyMind`, trained	63M	4,314	72.6

combines all worker outputs before one actor revision. Table 5 reports the comparison on the same five-dataset preliminary setting.

Table 5: Oracle Width, no-reflection Best-of-K, and naive aggregation on the five-dataset preliminary setting with LLaMA-3.1-8B. The width curves use one reflection cycle.

K	Oracle Width	Best-of-K, no reflection	Naive aggregation
1	64.0	64.0	64.0
2	70.2	66.9	66.8
4	76.4	69.5	68.0
8	81.9	71.8	74.2

From $K = 1$ to $K = 8$, Oracle Width increases by 17.9 points and no-reflection Best-of-K increases by 7.8 points. At $K = 8$, the difference between the two controls is 10.1 points. The oracle-naive gap is 7.7 points. Naive aggregation reaches 74.2 with one revision, 2.4 points above the eight-sample no-reflection control. `PolyMind` learns how to allocate and compress those signals.

The actor generates an initial answer and then one revision per reflection cycle. Naive aggregation and `PolyMind` combine worker feedback before each revision, while Oracle Width evaluates K separate reflection-conditioned revisions. Self-Consistency samples multiple complete answers.

The depth and width studies are independent scaling diagnostics: depth varies T with one worker, while width varies K in one reflection cycle. Appendix B compares methods under matched end-to-end token budgets.

D Training Details and Parametric Width Memory

Training data. Training uses 3,200 tasks: 1,200 code tasks from the MBPP training split and APPS introductory tasks [37], 1,200 mathematical tasks from GSM8K train and MATH train, and 800 multi-hop QA tasks from HotpotQA train and 2WikiMultiHopQA train. No test instance appears in training. The task mixture gives the reflection roles examples of different feedback formats while keeping the evaluation tasks separate.

Optimization and parameter sharing. The planner, worker, and reducer share one LLaMA-3.1-8B backbone with separate role-specific LoRA adapters. The actor is frozen and has no adapter. Table 6 summarizes the training configuration.

Parametric width memory. The record set $\mathcal{D}_{\text{width}}$ stores one triple per rollout: the task context, sampled width decision, and outcome reward. After each epoch, the memory update keeps the top 20% of its records by reward. The memory contains approximately 2,100 records. The lightweight module F_η is a two-layer MLP over a frozen 768-dimensional context embedding. It predicts a width in $\{1, \ldots, 8\}$ and a retrieval index over the 32 most frequent scope phrases among the selected records. The prediction is rendered as a short prior appended to the planner prompt. For example:

> Prior suggests $K = 4$ with scopes similar to {boundary conditions, loop indices, return type}.

Table 6: Training configuration for the default `PolyMind` setting.

Setting	Value
Training tasks	3,200
Rollouts per context, G	8
Epochs	2
Optimization steps	800
Total rollouts	Approximately 51,200
LoRA rank / alpha	16 / 32 per role
Learning rate	3×10^{-5}
Schedule / warmup	Cosine / 5%
Actor	Frozen, no adapter
Default $T_{\max}, K, L$	$2, 4, 256$
Hardware	8 NVIDIA A100 GPUs, 80GB each
Training cost	62 GPU-hours

The planner may follow or override this prior. It overrides the prior on 34% of instances; on those instances, its score is 3.2 points higher than forced adherence. Removing parametric width memory reduces the six-benchmark average from 72.6 to 71.2. The module provides a warm start while leaving the current task context available to the planner.

E Role-Balanced Learning and Scope Allocation

The role-balanced objective averages the worker contribution by K_g. Without this normalization, wider rollouts contribute more worker terms to optimization. Table 7 measures the resulting width-selection behavior.

Table 7: Effect of role-balanced normalization on width selection and six-benchmark performance.

Objective	Mean chosen K	Rollouts at $K = 8$	Avg.
With role-balanced loss	3.8	11%	72.6
Without role-balanced loss	7.6	91%	70.7

Without normalization, the planner favors maximum width and spends the budget on redundant workers. Role-balanced normalization reduces the frequency of maximum-width rollouts.

How scopes are generated. The planner starts from the pretrained backbone and is shaped by outcome reward through its LoRA adapter. It generates task-specific reflective scopes from the task context and evaluator feedback. Before training, near-duplicate scopes occur on 41% of instances with mean pairwise scope similarity 0.61. After training, these values are 12% and 0.28. The learned assignments therefore become more complementary.

A HumanEval example begins with an actor attempt that fails on an empty-list input. The planner chooses $K = 4$ and $L = 256$ and emits the following assignments:

1. **Boundary conditions:** inspect empty input, a single element, and maximum size. The output constraint is to list concrete failing inputs.
2. **Loop and index arithmetic:** inspect off-by-one errors at range endpoints. The output constraint is to trace one iteration explicitly.
3. **Return type and shape:** check consistency with the docstring signature. The output constraint is to state the expected and actual types.
4. **Input assumptions:** inspect assumptions about the input format that the problem does not state. The output constraint is to list each unstated assumption.

Each worker receives one assignment in isolation. The scope and its output constraint specify both what to inspect and how to make the resulting feedback actionable.

F Transfer Across Model Scales and Families

Table 8 evaluates depth saturation and width gains across four backbones, including the Qwen2.5 family [38]. The 8B and 7B settings train the reflection roles. The 32B and 70B settings use prompt-only `PolyMind` without role training.

Table 8: Transfer across scale and family. Gains and the $T = 5$ to $T = 10$ increment are in percentage points.

Backbone and setting	Best depth	Naive width	`PolyMind`	Gain	$T = 5 \rightarrow 10$
LLaMA-3.1-8B, trained	66.5	68.1	72.6	+6.1	+0.8
Qwen2.5-7B-Instruct, trained	68.9	70.4	74.8	+5.9	+1.1
Qwen2.5-32B-Instruct, prompt-only	75.6	76.8	80.1	+4.5	+0.6
LLaMA-3.1-70B, prompt-only	78.2	79.1	82.0	+3.8	+0.4

`PolyMind` improves over the best depth baselines by 6.1, 5.9, 4.5, and 3.8 percentage points across the four settings, respectively.

G Remaining Oracle Gap and Failure Cases

The remaining oracle gap depends on the type of correction required. Table 9 compares Oracle Width and `PolyMind` across four instance types at $K = 8$ in the five-dataset preliminary setting.

Table 9: Failure-type analysis at $K = 8$. Scores and closure rates are percentages.

Instance type	Oracle Width	`PolyMind`	Gap closed
Single localized defect	84.6	82.1	91%
Two or more independent defects	80.3	76.8	81%
Whole-approach change required	76.1	66.4	42%
Only one of eight workers finds the issue	79.5	68.9	47%

Gap closure is higher when a defect is local and several workers identify useful corrections. Two cases remain difficult. When the correct revision requires a different overall approach, the Keep field can anchor parts of an answer that should be discarded. When only one worker finds the real issue, the reducer can suppress that reflection as minority noise because it favors claims supported by several workers. These cases motivate per-claim confidence in addition to agreement among workers.

H Planner Prompt Excerpt

The following planner prompt core specifies the width and scope output format.

> You are allocating parallel review work on a failed attempt. Given the task, the current answer, and the evaluator feedback, output a JSON object with fields `K` (number of reviewers, 1 to 8), `L` (token cap per reviewer), and `scopes` (a list of K entries, each with `scope` and `constraint`). Each scope must name one distinct aspect of the current answer to inspect. Two scopes must not name the same aspect. Do not attempt to solve the task yourself.

I Additional Asset Sources and Licenses

The additional training tasks from APPS use the published training split [37]. Its official repository declares MIT. Qwen2.5-7B-Instruct and Qwen2.5-32B-Instruct [38] declare Apache-2.0 in their respective model cards [39, 40]. The Llama 3.1 family [41] is distributed under the Llama 3.1 Community License.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes].

   Justification: The claims concern reflective width allocation, structured compression, and empirical performance. The main results and the sampling, capacity, and transfer controls are reported in Section 5 and Appendices B, C, and F.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes].

   Justification: Appendix A discusses dependence on useful reflective feedback, additional inference cost, serving infrastructure, and the evaluated task domains. Appendix G discusses whole-approach changes and correct minority feedback.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete proof?

   Answer: [N/A].

   Justification: The paper is empirical and algorithmic and does not present formal theoretical results.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and conclusions of the paper?

   Answer: [No].

   Justification: Complete module and baseline prompts and implementation settings are not provided.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No].

   Justification: The experimental code and reproduction scripts are not publicly available.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details necessary to understand the results?

   Answer: [Yes].

   Justification: Section 5 specifies the datasets, metrics, baselines, default inference setting, and seed count. Appendices D, B, and C specify training data, optimization, parameter sharing, compute accounting, and the distinction between independent scaling diagnostics and matched comparisons.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No].

   Justification: The results are averaged over three seeds; per-seed dispersion and significance tests are not reported.

8. **Experiments compute resources**

Question: For each experiment, does the paper provide sufficient information on the computer resources needed to reproduce the experiments?

Answer: [No].

Justification: Serving configurations and per-experiment resource breakdowns are not fully specified.

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics?

   Answer: [Yes].

   Justification: The work studies reflection-based agents using public benchmarks and model checkpoints. It does not collect private data or involve human-subject research. Potential misuse is discussed in Appendix A.

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes].

    Justification: Appendix A discusses potential efficiency benefits and the misuse risks of more capable reasoning agents.

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse?

    Answer: [N/A].

    Justification: The paper does not release a new dataset, pretrained model, or high-risk asset.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets used in the paper properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [No].

    Justification: Licenses and terms of use are not specified for all assets.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A].

    Justification: The paper does not release a new dataset, benchmark, or model checkpoint.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation?

    Answer: [N/A].

    Justification: The paper does not involve crowdsourcing or human-subject experiments.

15. **Institutional review board approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board approvals or an equivalent approval were obtained?

    Answer: [N/A].

    Justification: The paper does not involve human-subject research, so IRB approval is not applicable.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research?

Answer: [Yes].

Justification: LLMs serve as the actor, planner, workers, and reducer. Their inputs, outputs, training roles, and parameter sharing are described in Sections 3 and 4 and Appendix D.