Tracing and Correcting Programs: Critic-Guided Synthesis for Visual Reasoning

Marha Midhatiey Rusli[1], **Donghyeon Shin**[1], **Sejin Kim**[1], **Sundong Kim**[1]

[1]Department of AI Convergence,
Gwangju Institute of Science and Technology,
Cheomdangwagi-ro 123, Gwangju, 61005, South Korea
marhamidhatiey@gmail.com, shindong97411@gmail.com, sjkim7822@gmail.com, sundong@gist.ac.kr

Abstract

Program synthesis for complex reasoning tasks faces a fundamental challenge: initial attempts often generate flawed programs that fail to capture the underlying problem logic. We introduce Tracing and Correcting Programs (TCP), a critic-guided framework that shifts the paradigm from *generate-and-test* to *critic-guided repair* through iterative refinement. Instead of discarding failed programs, TCP begins by analyzing each task, tracing its execution errors, and generating structured diagnostic feedback through a critic module. Through an iterative validation process, corrected programs are refined and tested until a solution emerges. Our key contributions include: (1) **A systematic approach that transforms failed synthesis attempts into improved and correct programs**, (2) **An adaptive sampling strategy** that allocates computational resources based on task complexity, requiring only 7–8 samples per task for complete solutions, and (3) **A zero-shot methodology that requires no task-specific training**. We evaluate TCP on the challenging Abstraction and Reasoning Corpus (ARC), which covers all 800 tasks, where TCP solves 159 tasks and improves the majority by up to 68.1%. Unlike evolutionary or multi-agent methods, which require evaluating hundreds or thousands of candidates, often with significant training overhead, TCP achieves systematic improvements with samples two orders of magnitude lower (300–400× reduction). These results highlight the importance of feedback-driven refinement and establish a new paradigm for efficient program synthesis in complex reasoning domains.

1 Introduction

Artificial intelligence systems are increasingly capable, yet solving problems that require abstract reasoning and systematic generalization remains a fundamental challenge. The *Abstraction and Reasoning Corpus* (ARC-AGI; hereafter ARC) (Chollet 2019) serves as a key benchmark demanding human-like inductive reasoning to infer hidden rules (e.g., symmetry, connectivity, object transformation) directly from a few visual input–output example pairs. ARC tasks (see Appendices for visualization of the task) are designed to test out-of-distribution generalization, often requiring the compositional application of simple operations (e.g., recoloring, moving, or resizing).

Recent years have seen growing interest in using program synthesis with Large Language Models (LLMs) to solve ARC, externalizing the underlying transformation logic as an interpretable program, moving beyond surface pattern matching. While powerful, LLMs' synthesis often fails due to subtle errors, such as incorrect object counting or improper operation sequencing. Existing methods (Balog et al. 2017; Bober-Irizar and Banerjee 2024) typically address this brittleness by exploring massive pools of candidate programs using guided search. However, the approach still depends on exploring large pools of candidates, with computational cost growing rapidly as program depth increases.

To overcome the inefficiency of extensive candidate enumeration, a growing body of work in LLM-based coding and reasoning has demonstrated the potential of *iterative repair*, where incorrect candidates are improved rather than discarded. For example in LDB (Zhong, Wang, and Shang 2024), ARCS (Bhattarai et al. 2025) and CodeIt (Butt et al. 2024). However, structured repair mechanisms remains rare in ARC. Current evolutionary (SOAR (Pourcel, Colas, and Oudeyer 2025)) and multi-agent strategies (MASR (Bikov, Bober-Irizar, and Banerjee 2025)), perform "blind" generations or reflections lacking structured diagnosis. This lack of fine-grained feedback risks making refinement as inefficient as unguided resampling.

Therefore, we propose *Tracing and Correcting Programs* (TCP). Our key contributions include: **(1)** A critic-guided synthesis framework that implements a structured, iterative repair loop guided by execution trace analysis. **(2)** Introduce a specialized Critic Module capable of generating both precise execution traces and high-level diagnostic feedback, which significantly boosts the LLM's ability to self-correct logical errors. **(3)** An adaptive selection mechanism that systematically determines the feedback (diagnostic granularity) and strategy level (actionable refinement guidance). Overall, TCP required an average of 2–3 iterations (which equals 7–8 samples per task) for complete solutions, demonstrating superior sample efficiency compared to contemporary program synthesis methods. The relations of how these components contribute to each other are illustrated in Figure 1.

These advancements establish a more effective and efficient paradigm for program synthesis in complex reasoning domains. We evaluate TCP by focusing on the framework's core benefits:

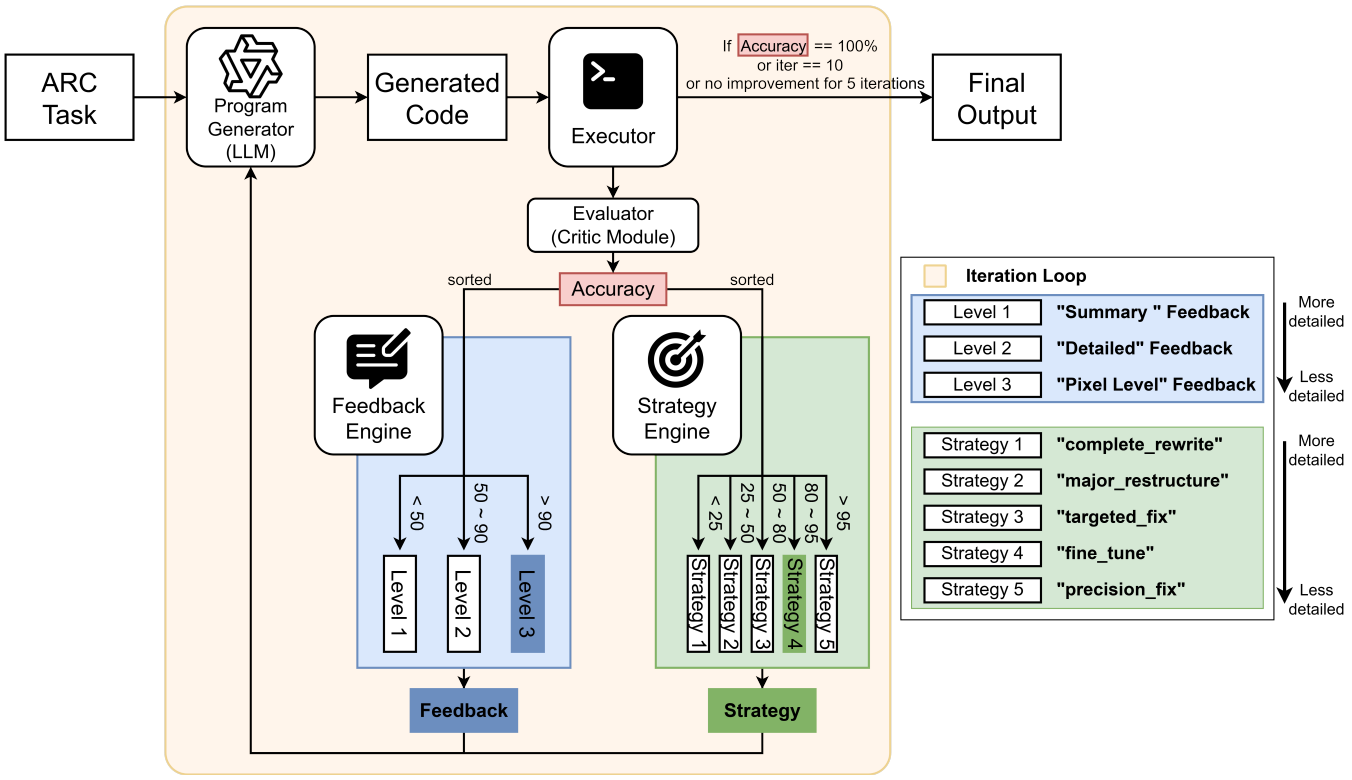


Figure 1: Overview of the TCP framework. An ARC task is provided as input–output pairs, is fed to the **Program Generator (LLM)**, which produces candidate Python code. This code is executed by the **Executor**, and the **Evaluator (Critic Module)** computes the pixel-wise accuracy and verdict checks, while enforcing a five-second execution limit. Crucially, these critic modules analyze the trace of a failed program and leverage adaptive selection to jointly determine one Feedback Level (either more detailed or less detailed) and one Strategy Level (either more detailed or less detailed). Programs are iteratively refined under this critic-guided loop until a correct solution is found or termination conditions are met. Our final configuration for the TCP loop, including the selection of iteration limits and other settings, was determined through extensive hyperparameter exploration. The complete details from this exploration are presented in the Appendices.

- **Performance:** Establish the performance of TCP in solving ARC tasks and improving failed tasks.
- **Efficiency:** Demonstrate that TCP provides superior sample efficiency, achieving systematic improvement with significantly fewer candidates per task compared to large-scale generate-and-test methods.
- **Effectiveness:** Isolate the contribution of the Critic module and adaptive feedback and strategy selection to validate the necessity of structured, dynamic correction.

2 Background

Evaluator (Critic Module) Once the Large Language Model (LLM) generates the programs based on the given input–output example pairs on an ARC task. This evaluator executes the programs within a 5-second timeout. The results from this evaluation are ***(1) Pixel-wise accuracy***. This is the primary metric denoted as α, which is defined as the proportion of correctly predicted pixels on the targeted pixels. This metric is crucial as it measures partial correctness, allowing the critic to generate structured feedback when the program fails the exact match requirement, where accuracy, $\alpha < 1$. Given an ARC task T consist of training pairs $\{(x_i, y_i)\}_{i=1}^{n}$, where $x_i, y_i \in \mathcal{G}$ are input and output grid instances, and test inputs $\{x_{\text{test}}\} \subseteq \mathcal{G}$, the goal is to learn a transformation function $f : \mathcal{G} \to \mathcal{G}$ that correctly maps all training inputs to their outputs and generalizes to test inputs. Let $\mathcal{G} = \{0, 1, \ldots, 9\}^{h \times w}$ denote the set of all $h \times w$ grids, where each cell contains a color from $\{0, \ldots, 9\}$.

A grid $y \in \mathbb{Z}^{h \times w}$ and its prediction $\hat{y}$, the pixel-wise accuracy is defined as:

$$\alpha(y, \hat{y}) = \frac{1}{h \times w} \sum_{i=1}^{h} \sum_{j=1}^{w} \mathbf{1}\left[y_{ij} = \hat{y}_{ij}\right]. \tag{1}$$

Here, h and w denote the height and width of the grid, respectively, and y_{ij} and $\hat{y}_{ij}$ denote the pixel values at position (i, j). ***(2) Verdict***. This is one of the components used for assigning the outcome, such as EQUAL (exact match), SHAPE MISMATCH, CONTENT MISMATCH, RUNTIME ERROR, or TIMEOUT. These two results (Pixel-wise accuracy and Verdict) served as the necessary signals for the Feedback & Strategy Level selection stage.

Strategy Engine Works as a selector on which Strategy Level $S(\alpha)$ to be chosen, either one between Level 1–Level 5 based on the pixel-wise accuracy.

$$S(\alpha) = \begin{cases} \text{“complete\_rewrite”} & \text{if } \alpha < 0.25 \\ \text{“major\_restructure”} & \text{if } 0.25 \leq \alpha < 0.50 \\ \text{“targeted\_fix”} & \text{if } 0.50 \leq \alpha < 0.80 \\ \text{“fine\_tune”} & \text{if } 0.80 \leq \alpha < 0.95 \\ \text{“precision\_fix”} & \text{if } 0.95 \leq \alpha < 1.00 \\ \text{“perfect”} & \text{if } \alpha = 1.00 \end{cases}$$

When $\alpha = 1.00$, refinement terminates. This strategy level determines the actionable refinement guide, which contains different components included in a prompt. Full details on the prompt are provided in the Appendices.

Feedback Engine This feedback engine also works as a selector on which Feedback Level $F(\alpha)$ to be chosen, either one between Level 1–Level 3 based on the pixel-wise accuracy. This dynamic feedback selection adapts to the selected feedback level, ranging from the most detailed to the least detailed. It comprises diagnostic components at iteration t:

$$F_t = \{\text{verdict}, \text{accuracy}, \text{color\_analysis}, \text{pixel\_details}, \text{pattern\_hints}\} \quad (2)$$

- **Level 1: Summary feedback** ($\alpha < 0.5$): High-level verdict(SHAPE MISMATCH), error category and accuracy score percentage.
- **Level 2: Detailed feedback** ($0.5 \leq \alpha < 0.9$): Detailed diagnostics—per-training-pair accuracy, missing and extra colors analysis, mismatch pixel counts.
- **Level 3: Pixel-level feedback** ($\alpha \geq 0.9$): Specific coordinate mismatches (up to 10 coordinates), expected vs. actual color values, Pattern-specific guidance.

Example “Level 1” feedback shown in Figure 2. At low accuracy, the feedback includes more detailed guidance, while at high accuracy, it receives less detailed guidance. This granular adaptation ensures the model receives appropriate guidance without overwhelming it with too much detail when the solution requires fundamental changes. The full detailed contents of each feedback level are shown in the Appendices.

Level 1 : Summary feedback

Content inserted into {final_feedback_payload}:

High-level verdict: SHAPE_MISMATCH
Accuracy: 23.4%
The code consistently fails with a SHAPE_MISMATCH error.
The output grid dimensions are incorrect.

CRITICAL: Very low accuracy. The core transformation logic is wrong.

Figure 2: The example of Level 1 “Summary” feedback.

Pattern Analysis Implementation for Pattern & Structural Hints Generation The hints provide essential context that helps the model understand the transformation requirements before attempting code generation or refinement, significantly improving the quality and relevance of generated solutions. The system analyzes the input–output example pairs to extract structural transformation properties that can guide the refinement process based on its pattern type, as shown in Table 1. These properties are converted into contextual hints to be included in the feedback level contents and strategy level prompts. ***(1) Grid Dimension Analysis:*** which compares input and output grid shapes to determine whether the dimensions are preserved (sizes match) or change (resizing occurs). ***(2) Color Usage Analysis:*** that extracts the color set from both grids to identify any color additions, removals, or substitutions in the output.

3 Methods

TCP Framework Overview

The TCP framework operates as an iterative champion–challenger loop. Given an ARC task (input–output example pairs), the system attempts to find a correct transformation f within a limited number of refinement steps.

TCP operates in four stages:

- **Program Generator**: The LLM generates the initial and candidate programs in Python code based on task input–output examples.
- **Evaluator (Critic Module)**: The generated program is executed on the training pairs. Execution is subject to a 5-second timeout. The Evaluator measures pixel-wise accuracy α and assigns categorical verdicts (e.g., EQUAL, SHAPE MISMATCH). This is defined as the proportion of grid cells where the predicted value matches the target value. Pixel-wise accuracy metric is computed as in Eq. (1). If 5 out of 9 cells are correct, the pixel accuracy is 56%. This measure captures partial correctness even when the entire grid is not a perfect match. The evaluator returns the verdict and the pixel-wise accuracy scores as the necessary signals to determine the Feedback and Strategy Level.
- **Feedback and Strategy Selection**: If accuracy $\alpha < 1$, the critic module diagnoses the failure by analyzing execution traces. It first explores the transformation patterns of the task (e.g., ’Color Mismatch’, ’Grid Misalignment’) and generates hints based on it. The complete set of heuristic rules and their corresponding hints are detailed as in the *Background* section. This diagnosis will be included within the chosen Feedback and Strategy Level, and will make the refinement based on the hints of the transformation patterns.
- **Refiner (Champion–Challenger Loop)**: The LLM is reprompted with the faulty code, structured diagnostics, and the selected strategy to generate an improved candidate. The new candidate only replaces the current best program (the **champion**) if its accuracy, α_n (the accuracy of the candidate n in the current iteration), strictly improves upon the champion’s accuracy ($\alpha_n > \alpha_{\text{champion}}$). Figure 3 and 4 show how this champion update process works.

Pattern Type	Description	Generated Hint
Dimension Preservation	Grid size remains constant	Grid dimensions are preserved
Grid Transformation	Grid resizing or cropping	Grid dimensions change between input and output
Color Substitution	Direct color replacement (e.g., all 3s become 7s)	Color mapping detected: $3 \rightarrow 7$, $1 \rightarrow 4$
Color Addition	New colors appear in output	New colors appear: $\{4, 7, 9\}$
Color Removal	Input colors missing from output	Colors removed: $\{2, 5\}$
Color Preservation	Same color maintained	Color preserved

Table 1: Transformation pattern classification and generated hints. The system categorizes transformations into common ARC pattern types.

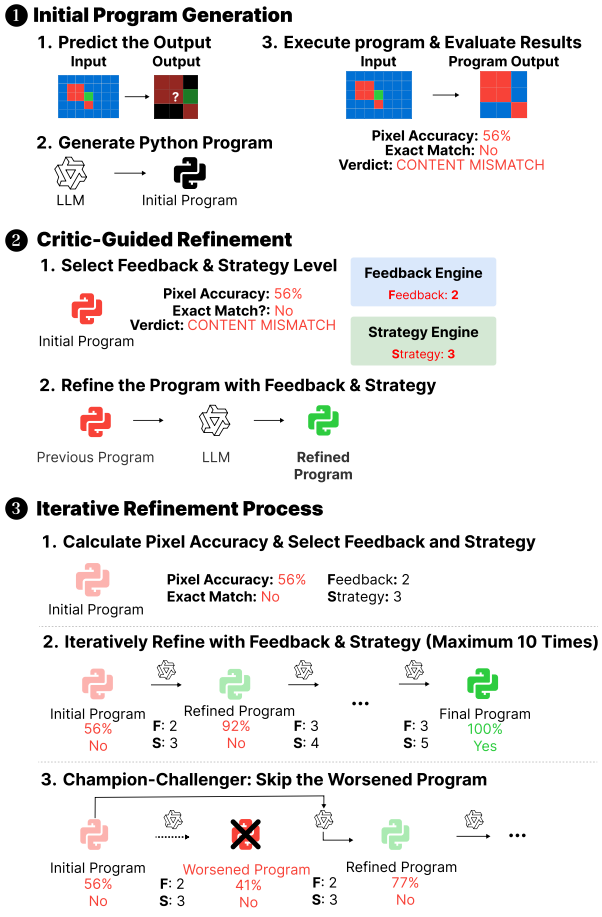


Figure 3: Whole process of TCP. (1) **Initial Program Generation:** Given input–output examples, the LLM predicts an output, generates a Python program, and executes it to obtain results. (2) **Critic-Guided Refinement:** The critic evaluates execution with pixel-level accuracy and verdict checks, determines a feedback and strategy level (e.g., rewrite, restructure, fix, etc.), and produces structured diagnostics. (3) **Iterative Refinement Process:** Programs are refined up to 10 times, with improved candidate programs validated against the target output. A champion–challenger mechanism skips worsened programs, ensuring only stronger candidates survive. The example progression shows accuracy improving from 57% to 100%.

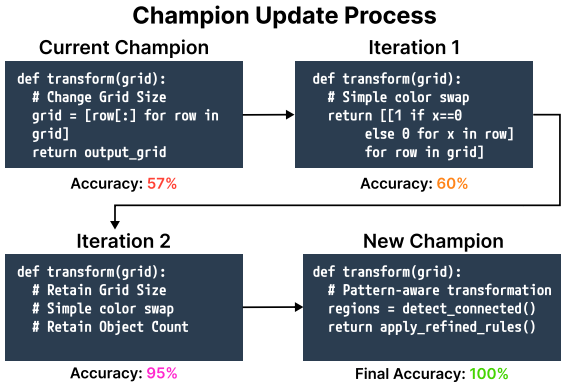


Figure 4: Demonstration of the champion update process. As accuracy improves through iterative critic-guided refinement, the program with the highest accuracy becomes the new champion.

Adaptive Selection and Feedback Generation

The core contribution of TCP is its dynamic control mechanism, which prevents inefficient random regeneration. This control is governed by the adaptive selection, which jointly determines the required diagnostic details (**Feedback Level,** F) and the appropriate corrective action (**Strategy Level,** S) based on the pixel-wise accuracy α. This joint determination balances the computational cost of detailed diagnostics against the necessity of major code revision. This entire decision logic, as formalized in Table 2, balances the need for a more detailed strategy and feedback (e.g., *complete_rewrite, summary level*) at low accuracy and a less-detailed approach (e.g., *precision_fix, pixel level*) at high accuracy. The rationale for these thresholds is empirically tuned: low accuracy necessitates fundamental changes, while high accuracy allows for the efficient application of minimal corrections guided by granular feedback.

Algorithm

The entire refinement process is formalized in Algorithm 1, which is presented in the Appendices for detailed review. All accuracy (α) and signals used (F and S) in TCP are strictly applied on training pairs. Test outputs are never used during generation or refinement; they are only accessed once at the final evaluation, consistent with the ARC-AGI protocol.

Feedback level (F)	Content	Strategy Level (S)	Refinement action
Level 1 ($\alpha < 0.5$)	Summary verdict (e.g., SHAPE MISMATCH); accuracy score only	*Complete rewrite*	Discard the current code and regenerate from scratch.
Level 2 ($0.5 \leq \alpha < 0.9$)	Per-training-pair accuracy; color and object mismatch analysis; pixel count comparison	*Major restructure / Targeted fix*	Revise program structure or correct localized logical errors.
Level 3 ($\alpha \geq 0.9$)	Pixel-level mismatches (up to 10 coordinates, values) with pattern-specific hints	*Fine-tune / Precision fix*	Apply minimal fix to achieve exact solution (e.g., "Position [6,7]: expected 4, got 2").

Table 2: Adaptive selection module: the criteria for joint determination of feedback and strategy. This table governs the critic's action based on program execution results.

Temperature Adaptation

This temperature adaptation follows a strategic exploration-exploitation approach for iterative code repair. When generating the newly refined programs in the early iterations ($i < 3$), we want it to explore a more diverse approach as a "warm start." Once the solution is nearly perfect ($\alpha > 0.85$), it requires small and deterministic fixes, as high temperatures will introduce random changes that break the working code. When the core logic is almost correct ($\alpha > 0.70$), some exploration is needed, but it should be more focused than in the early stages, to avoid breaking what is working while addressing the remaining issues. At low accuracy, standard exploration with moderate temperature is needed. This is decided based on the iteration i and accuracy α, starting from a base temperature $\tau_{\text{base}} = 0.4$:

$$\tau(i, \alpha) = \begin{cases} \min(0.7, \tau_{\text{base}} \times 1.75) & \text{if } i < 3 \\ \max(0.15, \tau_{\text{base}} \times 0.4) & \text{if } \alpha > 0.85 \\ \max(0.25, \tau_{\text{base}} \times 0.6) & \text{if } \alpha > 0.70 \\ \tau_{\text{base}} & \text{otherwise} \end{cases}$$

Best-of-N Sampling and Hypothesis Tracking

Best-of-N Sampling For each strategy, TCP generates N solution candidates per iteration. Each candidate is evaluated independently, with the best-performing solution becoming the new champion if it improves upon the current accuracy:

$$N(S) = \begin{cases} 5 & \text{if } S = \text{"precision\_fix"} \\ 4 & \text{if } S = \text{"complete\_rewrite"} \\ 3 & \text{if } S = \text{"major\_restructure"} \\ 2 & \text{otherwise} \end{cases}$$

Hypothesis Tracking. TCP maintains a hypothesis history $H = \{h_1, h_2, \ldots, h_t\}$ to prevent repetition and guide exploration. Each refinement prompt P_{refine} contains structured elements that preserve the task context and diagnostic history, including `task_description`, `training_examples`, `current_code`, `current_accuracy`, execution feedback (F_t), the three most recent hypotheses $H[-3:]$, and `pattern_hints`.

The LLM then returns a structured refinement consisting of four components: (1) `<pattern_analysis>` describing color or shape transformations, (2) `<hypothesis>` proposing a new conceptual rule distinct from prior attempts, (3) `<plan>` outlining the corrective logic, and (4) an updated `transform()` implementation. This structured reasoning makes the refinement process both interpretable and verifiable.

4 Results

Experimental Setup

We utilized the public portion of the Abstraction and Reasoning Corpus (ARC-AGI-1), which comprises 800 tasks (400 training sets and 400 evaluation sets). All experiments were performed on three model sizes from the Qwen2.5-Coder family (7B, 14B, and 32B Instruct models) (Hui et al. 2024). The precise hardware specifications (GPU type and execution limits) and LLM model variants used (7B, 14B, and 32B) are listed in the Appendices. All programs were executed within a 5-second timeout per run. Task inputs were limited to a 30×30 grid size. We employ four primary evaluation metrics:

1. **Solved tasks**: The percentage of tasks where TCP generates a perfectly correct program ($\alpha = 1.0$) (exact match on training and test examples).
2. **Improved tasks**: The percentage of final program accuracy ($\alpha_{initial} < 1.0$) higher than the initial candidate after I_{max} iterations ($\alpha_{\text{final}} > \alpha_{\text{initial}}$) but not an exact solution.
3. **Failed tasks**: No improvement from the initial accuracy
4. **Final accuracy:** average pass@1 accuracy on the validation pair for each task, i.e., only the chosen candidate solution from the training is considered.
5. **Iteration efficiency**: The total number of iteration steps generated per task required to achieve the final accuracy.

Effectiveness of TCP: Solve and Improvement Rates across LLM models

We first establish TCP's capability to solve and improve ARC tasks. As shown in Table 3, TCP consistently demon-

Model	Solved	Improved	Failed
Qwen2.5-Coder-7B-Instruct	47	473	280
Qwen2.5-Coder-14B-Instruct	77	468	255
Qwen2.5-Coder-32B-Instruct	126	417	257

Table 3: Overall performance of TCP on ARC tasks on different models. The table reports the number of solved, improved, and failed tasks. This table highlights that TCP produces systematic improvement across backbones while solve counts increase with model scale.

strates a high number of improved functions across all models, confirming the general efficacy of the Critic-Guided Refinement Loop. Notably, the high improvement rates, which reach up to 68.1%, confirm that the framework successfully repairs a vast majority of initially flawed programs. The scaling results show that the core mechanism is model-agnostic and scales effectively with larger, more powerful LLM backbones, achieving the highest overall solve rate with the Qwen-32B model. This robustness confirms that TCP's systematic guidance successfully utilizes the increased reasoning capacity of larger models. Full results on how TCP performance scales with different backbone sizes are included in the Appendices.

Comparison with Baselines: Efficiency Performance Tradeoff vs. Prior ARC Solvers

We compare TCP against established ARC solvers, such as DreamCoder, MASR, and SOAR, as shown in Table 4. While advanced search techniques like SOAR (Pourcel, Colas, and Oudeyer 2025) and MASR (Bikov, Bober-Irizar, and Banerjee 2025) may achieve higher overall solve rates through extensive search, TCP offers a compelling trade-off. TCP is competitive in solving rate, but achieves significantly lower resource usage, as evidenced by its superior sample efficiency metrics (explained in *Sample Efficiency* results). This confirms that TCP successfully positions itself as an efficient alternative in the reasoning tasks, prioritizing intelligent resource use through structured critique over large-scale and random "blind" generation.

The efficiency performance graph in Figure 5 shows TCP's advantage over prior ARC methods. Compared to prior solvers, TCP trades raw solve counts for efficiency. This makes TCP an efficiency-first synthesis framework, offering a scalable alternative to methods with large candidate pools. Methods like SOAR require more samples per task to achieve higher solve counts, whereas DreamCoder falls into the dominated region with poor efficiency. Instead of using more resources, TCP introduces targeted refinement loops that yield broad partial improvements efficiently. This makes TCP complementary rather than competitive; it can integrate with existing search-based systems to reduce candidate evaluations while maintaining accuracy. As an efficiency-first framework, it serves as a lightweight refinement module that amplifies existing solvers. Additionally, TCP explicitly reports partial progress by distinguishing solved from improved ones, better reflecting the reasoning ability on ARC,

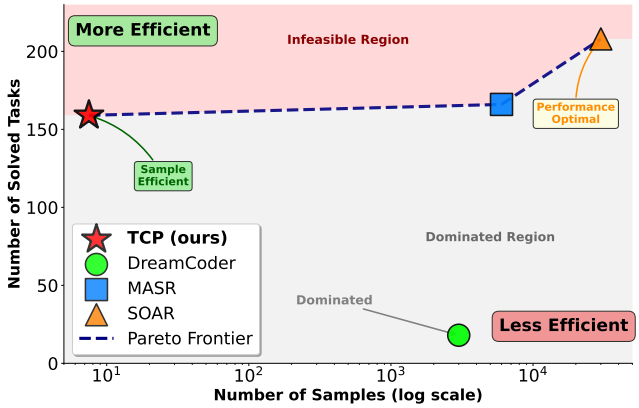


Figure 5: Efficiency-performance trade-offs: TCP vs. prior ARC methods.

where incremental improvements represent meaningful abstraction skills. The Pareto frontier demonstrates that TCP opens new efficiency methods previously considered infeasible, making it valuable for budget-constrained scenarios.

Sample Efficiency: Iterations and Candidate Evaluations to Convergence

A key hypothesis of this work is that the trace-and-correct approach is more sample-efficient than large-scale generate-and-test methods, as shown in Figure 6. The results show a stark contrast in resource consumption. TCP achieves its final performance within an average of only 2–3 iteration steps (7–8 candidates generated per task since each iteration generates three candidate samples), drastically outperforming baselines that require hundreds or even thousands of samples to achieve comparable accuracy. The low number of iteration steps per solved task demonstrates that TCP successfully guides the LLM toward convergence with minimal generation steps, validating the efficiency claim of the trace-and-correct paradigm.

The sharp decline from Iteration 1 (800 tasks) to Iteration 2 (330 tasks) shows TCP's ability to quickly solve or improve a large proportion of tasks in the first attempt. The gradual flattening of this curve after Iteration 3-4 indicates that remaining tasks require progressively more iterations, suggesting that these are more difficult problems that require multiple refinement cycles. This pool-decreasing pattern is important for interpreting the bar heights: the decrease in task counts at later iterations is due to the shrinking pool size, not solely to TCP's reduced effectiveness. For example, in Iteration 9, only 2 improved tasks are shown, which is a result of a remaining pool of just 33 tasks, compared to the 800 tasks available at Iteration 1.

This illustrates that TCP achieves a reduction of up to two orders of magnitude in sample efficiency while still producing meaningful improvements, as shown in Eq. (3). Importantly, the refinement dynamics also differ: TCP progresses step by step on a single candidate, whereas other approaches discard weaker attempts. This repair-oriented strategy reflects a more human-like debugging process, where partial

Method	ARC Correct (Task)	Avg. Samples	Strategy
DreamCoder (Bober-Irizar and Banerjee 2024)	88	∼3000	Neuro-symbolic
MASR (Bikov, Bober-Irizar, and Banerjee 2025)	163	2000 w/ QLoRA Fine-Tuning	Multi-agent reflection
SOAR (Pourcel, Colas, and Oudeyer 2025)	208	6000 × 5 / Task	Evolutionary mutation
TCP (ours)	159	**∼7–8 / Task**	Critic-guided refinement

Table 4: Comparison with prior ARC methods. TCP trades solve count for efficiency and systematic refinement.

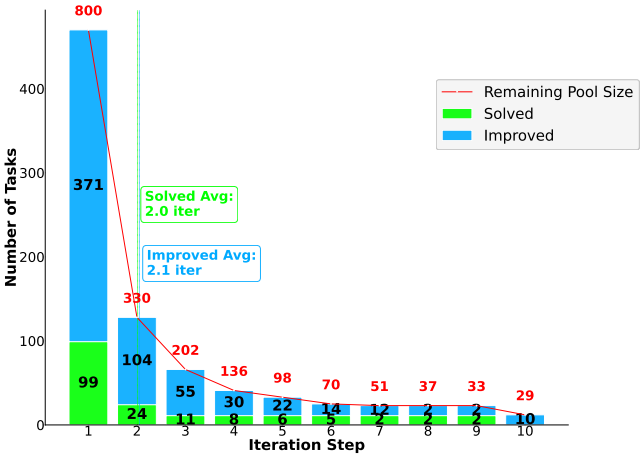


Figure 6: Distribution of iterations required to Solved or Improved tasks with TCP. Each bar represents the number of unique tasks completed at a given iteration across all Qwen2.5-Coder-Instruct models (green = solved, blue = improved). The red line shows the remaining unsolved task pool, illustrating cumulative task completion. Averages reported in the text (Solved Avg = 2.0 iter, Improved Avg = 2.1 iter) are computed across all tasks. The total number of solved tasks (123) converges within only 2 iterations, confirming TCP's sample efficiency.

correctness is preserved and gradually built upon rather than resetting at each attempt. Formally, the computational cost reduction can be expressed as:

$$\frac{\text{Cost}_{\text{baseline(DreamCoder=3000)}}}{\text{Cost}_{\text{TCP(7 to 8)}}} \approx \begin{cases} 429\times, & \text{for 7 samples} \\ 375\times, & \text{for 8 samples} \end{cases} \tag{3}$$

showing that TCP achieves comparable improvements with up to two orders of magnitude more efficiency than large-scale methods (e.g., DreamCoder (Bober-Irizar and Banerjee 2024)), proving the value of targeted refinement.

Ablation Study: Critic Feedback's Contribution

To assess the effect of the critic's feedback, we conducted an ablation study comparing the TCP against two critical variants: a TCP with no-critic, which refines programs without any structured feedback, and a TCP static. The static critic was designed to isolate the effect of adaptivity by providing only a fixed "detailed" feedback (Level 2) and consistently applying a "major restructure" strategy (Level 3), regardless of the program's current accuracy. This design ensures all other pipeline elements, such as sampling counts, temperature selection, and termination rules, are constant.

As shown in Table 5, the critic enabled TCP to solve 43–114 tasks. Most notably, the **TCP static only solved 28–93 tasks**, while **no-critic baseline achieved zero solved tasks** across all models. This result may appear extreme, but it is consistent with recent findings by (Pourcel, Colas, and Oudeyer 2025), where state-of-the-art models such as Claude-3.5, Claude-4, and GPT-4.1 achieve only 8–20 solved tasks in one-shot sampling, and open-source Qwen-72B or Mistral-large achieve less than five solved tasks. Additionally, TCP with critic also substantially increases the number of improved tasks: for Qwen2.5-Coder-7B-Instruct, critic feedback enables improvement in 196 tasks compared to 149 tasks without critic. These results highlight the fragility of raw generation and confirm that structured critic feedback is essential for efficient program repair. The complete ablation results, including the distribution of solved, improved, and failed tasks for each variant, are presented in Figure 7.

5 Discussion

Interpretation of Key Results

The experimental evidence strongly validates the core hypothesis of TCP: that critic-guided refinement is significantly more resource-efficient than large-scale generation for complex visual reasoning tasks, such as ARC.

- **Efficiency over blind generation.** Results in the *Sample Efficiency* show that adaptive coupling of feedback granularity and strategy level minimizes wasted LLM calls. Targeted diagnostics allow the model to bypass open-ended exploration and converge with few attempts, which is critical for scalable program synthesis.
- **Necessity of the critic.** The ablation study in the *Ablation Study* reveals substantial performance drops when either dynamic or static feedback is removed. This confirms that

Model	Variant	Solved	Improved
Qwen2.5-Coder 7B-Instruct	w/o Critic	0	149
	TCP static	28	201
	TCP with Critic	43	196
Qwen2.5-Coder 14B-Instruct	w/o Critic	0	102
	TCP static	54	178
	TCP with Critic	69	173
Qwen2.5-Coder 32B-Instruct	w/o Critic	0	145
	TCP static	93	143
	TCP with Critic	114	134

Table 5: Ablation study on 400 tasks across different Qwen2.5-Coder-Instruct models. Removing the critic module significantly reduces performance, confirming its critical role in TCP.

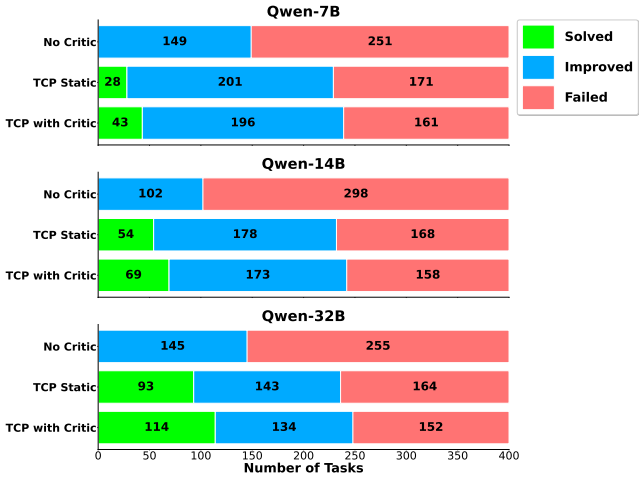


Figure 7: Full Ablation Study on 400 tasks. It clearly illustrates how removing the critic feedback signals leads to a lower number of solved tasks and a higher proportion of failed tasks, highlighting the critic's crucial role in guiding effective refinement.

debugging signal quality matters more than the quantity of generated candidates.

- **Generalizability and Scaling**: The high improvement rates across the 7B, 14B, and 32B Qwen models in *Effectiveness of TCP* analysis demonstrate the framework's model-agnostic nature. TCP effectively translates the enhanced capacity of larger LLMs into improved outcomes by providing a reliable mechanism to leverage that capacity for structured correction.

Broader Implications

The success of TCP highlights the potential of explicit neuro-symbolic integration for generalization tasks. The framework partitions the problem: the LLM (the "neuro" part) handles hypothesis generation and code execution, while the deterministic critic (the "symbolic" part) provides the rigorous, trace-and-correct verification and diagnosis.

- **Model Governance**: TCP's champion-challenger dynamic, adapted from risk modeling (Siddiqi 2012; Kuhn, Johnson et al. 2013), proves effective for governing LLM outputs. This ensures monotonic accuracy improvement, preventing oscillation and guarding against regressions common in open-ended LLM refinement.
- **Future of LLM Reasoning**: This work suggests a path forward for LLMs in complex domains: moving from being mere generators to becoming efficient debuggers capable of critically analyzing their own code and reasoning traces. The systematic approach to error correction introduced by TCP can be applied to other domains where deterministic execution and structured feedback are available (e.g., formal proofs, database queries, robotic planning).

Limitations and Future Work

Efficiency vs. Solve Rate While TCP solved 159 tasks, fewer than SOAR (Pourcel, Colas, and Oudeyer 2025) (208 tasks) or MASR (Bikov, Bober-Irizar, and Banerjee 2025) (166 tasks), it achieves this under a far stricter budget: one A100 GPU, mid-scale Qwen models, and only a few minutes per task. In contrast, many baselines rely on closed-source LLMs (such as GPT-4 and Claude) or hundreds of thousands of candidates, often utilizing high-end GPU. TCP, therefore, introduces a complementary axis of progress: efficiency and systematic refinement, highlighting advancement beyond raw accuracy. These paradigms outline a spectrum of strategies—accuracy maximization through scale on one end, and efficiency and interpretability on the other. Future models may leverage both scaling and critic-guided repair.

The Role of Improvements We explicitly report "improved" tasks—cases where partial correctness is achieved but the final program is not exact—because ARC is designed to test reasoning, not just binary success. For example, programs that get color mappings right but have misplaced shapes demonstrate real progress in abstraction. TCP's ability to systematically turn failed candidates into improved ones highlights a reasoning path that large-scale generate-and-discard methods cannot capture. Furthermore, these incremental improvements provide a natural learning signal for future work on adaptive or learned critics, where the critic itself can be refined based on patterns of partial success.

6 Conclusions

This work introduced **Tracing and Correcting Programs (TCP)**. This critic-guided Synthesis framework dramatically improves the sample efficiency of LLM-based solutions for the ARC benchmark in visual reasoning tasks. By dynamically coupling trace-and-correct error diagnosis with adaptive refinement strategies, TCP demonstrated superior efficiency and robust performance across multiple LLM backbones. TCP validates the "trace-and-correct" paradigm, establishing a resource-responsible method for advancing reasoning in complex, systematic domains.

A Algorithm

Algorithm 1: Critic-Guided Refinement (TCP Loop)

Input: Task T with training pairs $\{(x_i, y_i)\}$, test inputs $\{x_{\text{test}}\}$
Output: Transform function f
Phase 1: Initial Generation
$P_{\text{init}} \leftarrow \text{ConstructInitialPrompt}(T)$
$code_0 \leftarrow \text{GenerateCode}(P_{\text{init}}, \tau = 0.7)$
$(\alpha_0, F_0) \leftarrow \text{Evaluate}(code_0, \{(x_i, y_i)\})$
// α: pixel accuracy on training pairs, F: verdict
Phase 2: Iterative Refinement
$champion \leftarrow code_0$
$\alpha_{\text{best}} \leftarrow \alpha_0$
$H \leftarrow \emptyset$ *// hypothesis history*
for $t = 1$ **to** $I_{\max}$ **do**
  $S \leftarrow \text{SelectStrategy}(\alpha_{\text{best}})$
  if $S =$ "perfect" **then**
    break
  end if
  $\tau \leftarrow \text{AdaptTemperature}(t, \alpha_{\text{best}})$
  $N \leftarrow \text{DetermineSamples}(S)$
  $F \leftarrow \text{GenerateFeedback}(\alpha_{\text{best}}, \text{level} = \text{GetDetailLevel}(\alpha_{\text{best}}))$
  $candidates \leftarrow \emptyset$
  for $n = 1$ **to** N **do**
    $P_{\text{refine}} \leftarrow \text{ConstructRefinementPrompt}(T, champion, F, H, S)$
    $code_n \leftarrow \text{GenerateCode}(P_{\text{refine}}, \tau)$
    $h_n \leftarrow \text{ExtractHypothesis}(code_n)$
    $(\alpha_n, F_n) \leftarrow \text{Evaluate}(code_n, \{(x_i, y_i)\})$
    $candidates \leftarrow candidates \cup \{(code_n, \alpha_n, h_n)\}$
  end for
  $challenger \leftarrow \arg\max_{c \in candidates} c.\text{accuracy}$
  if $challenger.\alpha > \alpha_{\text{best}}$ **then**
    $champion \leftarrow challenger.\text{code}$
    $\alpha_{\text{best}} \leftarrow challenger.\alpha$
    $H \leftarrow H \cup \{challenger.\text{hypothesis}\}$
  end if
  if NoImprovement(last_5_iterations) **then**
    break
  end if
end for
Final Evaluation on Unseen Test Inputs
Run $champion$ on $\{x_{\text{test}}\}$ to produce outputs
return $champion$, outputs

B Detailed Experimental Setup

B.1 Model Configuration and Resources

All experiments were conducted using three variants of the Qwen2.5-Coder family (7B, 14B, and 32B Instruct models) on a single NVIDIA A100-80GB GPU. Each model operated with a context window of 30,000 tokens and a maximum generation length of 2,048, 3,072, or 4,096 tokens, depending on model size. The average GPU memory utilization was approximately 85%.

B.2 Parameter Exploration for Temperature Adaptation and Iteration Step

To evaluate the effect of adaptive Temperature, Seed, and Iteration settings, an experiment is conducted across 3, 5, and 7 iterations with varying temperature values (0.4, 0.5, 0.6) for each iteration number. Each configuration was tested on a set of 5 tasks to measure improvement in accuracy.

Iteration	Temp	Seed	Improved / 5	Avg Gain	Tier
7	**0.4**	**42**	**5 / 5**	**0.552**	**Top**
5	0.6	123	5 / 5	0.548	Top
7	0.4	123	5 / 5	0.546	Top
7	0.6	0	5 / 5	0.542	Top
5	0.4	42	4 / 5	0.525	Mid
3	0.4	123	4 / 5	0.545	Mid
3	0.6	123	3 / 5	0.550	Lower
3	0.4	42	3 / 5	0.547	Lower

Table 6: Parameter exploration across iterations, temperatures, and seeds. The best configurations consistently improved all five tasks.

The primary performance metric is the number of tasks that show improvement, reflecting both the consistency and effectiveness of a given configuration. The optimal configurations are those that achieve improvement across all 5 tasks, with top-tier performance observed in the 7-iteration runs at temperatures of 0.4 or 0.6.

The best-performing configuration was observed at **Iteration 7**, with a **temperature of 0.4** and **seed 42** (green row in Table 6). Under this setting, all five tasks showed improvement, achieving the highest average accuracy gain of **0.55%**. Other top-tier configurations, such as **Iteration 5, Temperature 0.6, Seed 123** and **Iteration 7, Temperature 0.4, Seed 123**, also consistently improved all tasks and yielded comparably high accuracy gains. Overall, higher iteration counts (five or seven) generally produced more stable improvements than three iterations. Temperature values in the range of **0.4–0.6** proved optimal, whereas extreme values (0.0 or 1.0) tended to degrade performance. Finally, the observed variability across random seeds indicates that stochasticity plays a notable role in influencing the final outcomes.

C Full and Extended Results

C.1 Computational Time Distribution for Solved Task

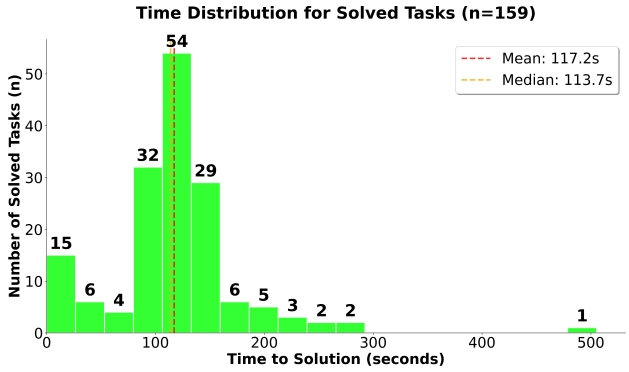
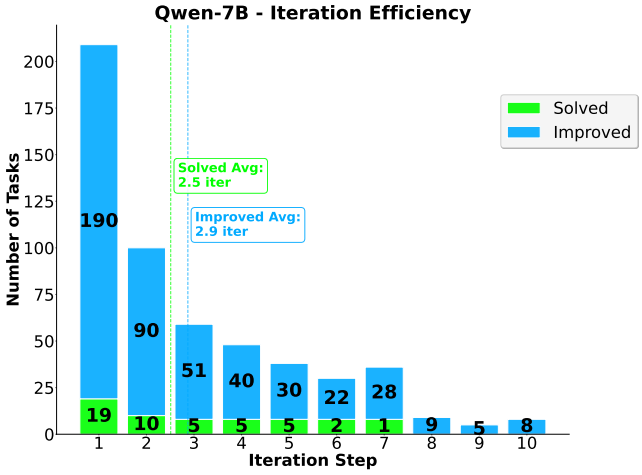


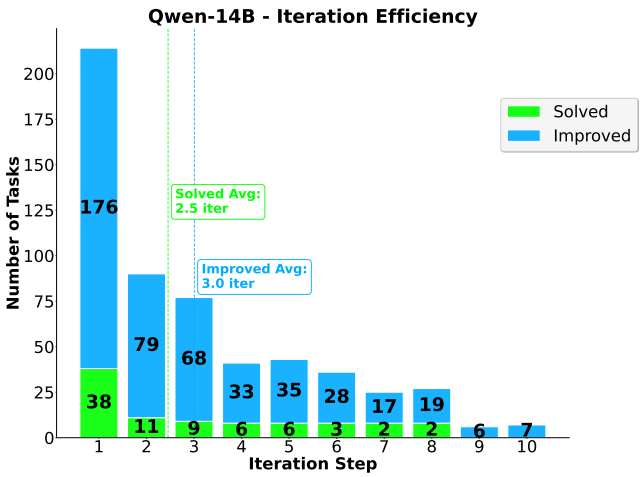
Figure 8: Time Distribution for Solved Tasks (n=159). The count of tasks solved within specific time bins is displayed, showing that the majority of tasks (54) were solved between approximately 100 seconds and 150 seconds. The dotted lines in red show that the average time to solve a task was 117.2 seconds.

Figure 8 explains how the time taken to solve 159 tasks is distributed. The reason for showing this graph is to illustrate that most of the tasks were solved relatively quickly, with 54 tasks solved between 100–150 seconds, and 15 tasks that were solved between 0–25 seconds, suggesting two distinct classes of tasks: easy tasks that were solved almost immediately, and difficult tasks that require more time. Beyond this peak, tasks that take more than 200 seconds become rare, indicating that a relatively small number of tasks account for the majority of the solution time. This demonstrates that our method has relatively consistent performance characteristics.

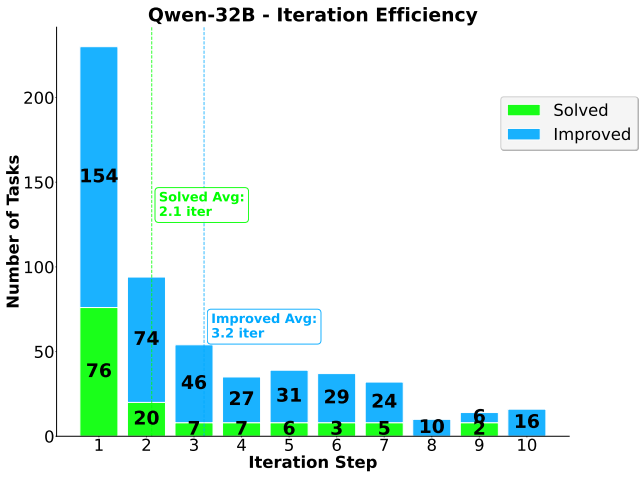
C.2 Backbone Scaling: TCP Scaling results across different Model Sizes



(a) Qwen-7B



(b) Qwen-14B



(c) Qwen-32B

Figure 9: TCP scaling results across different LLM models. (a) Qwen-7B, (b) Qwen-14B, (c) Qwen-32B.

Figure 9 shows a clear monotonic trend: as the model size increases from 7B to 32B, more tasks are solved while iteration counts remain stable, indicating more reliable convergence with larger backbones. Larger models such as Qwen2.5-Coder-14B-Instruct and Qwen2.5-Coder-32B-Instruct solve more tasks and converge in fewer iterations than Qwen2.5-Coder-7B-Instruct.

D Full Prompt Templates

D.1 Prompt Construction

The initial prompt P_{init} comprises four main components to ensure a consistent input format and immediate executability for all candidate programs. It includes the **(i) task specification with color mappings**, followed by the **(ii) training pairs** represented as NumPy arrays annotated with their respective dimensions. The prompt also defines the **(iii) function signature** as: `transform(input_grid: list[list[int]]) -> list[list[int]]`, which specifies the expected input–output format. Finally, an **(iv) optional one-shot example**, randomly selected from previously successful solutions, can be included to provide the model with an additional reference for generalization.

D.2 Strategy Level Prompt

These are the complete prompt templates for each of the five Strategy Levels as shown in Figures 10, 11, 12, 13, and 14. These are the core mechanism that guides the Large Language Model (LLM).

D.3 Feedback Level Contents Details

The full details of the contents for the remaining two over three levels of Feedback are shown in Figures 15 and 16.

E ARC Task Example Visualization

This is visualization on few different ARC(Abstraction and Reasoning Corpus) tasks looks like as shown in Figure 17.

"complete_rewrite" (α < 0.25)

You are an AI assistant specialized in solving ARC-AGI tasks.
CRITICAL: The current solution has only {champion_accuracy:.1%} accuracy after {i+1} attempts.
This indicates the approach is FUNDAMENTALLY WRONG.

Task to solve:
{item['problem_description']}

Pattern Analysis:
{structural_hints}

Training Examples:
{training_examples_str}

FAILED APPROACH (DO NOT COPY THIS):
```python
**{champion_code}**

# Feedback showing why it fails:

**{final_feedback_payload}**

# Previously tried hypotheses (AVOID REPEATING THESE):

**{previous_hypotheses}**

Your task is to act as an expert problem solver. Follow these steps:

1. Analyze the Pattern: Inside <pattern_analysis>
tags, describe what actually changes between input
and output grids. Look for:
  - Object detection and manipulation
  - Color replacement rules
  - Pattern filling or flood fill
  - Symmetry operations
  - Counting and replication
2. Formulate a NEW Hypothesis: Inside <hypothesis> tags,
   describe a COMPLETELY NEW approach that is different
   from the failed code above and different from previously tried hypotheses.
3. Plan the Implementation: Inside <plan> tags, describe how you will implement
   this new hypothesis step by step.
4. Write the Code: Provide a complete NEW implementation of the transform
    function.

Remember: The current approach is fundamentally wrong. Start fresh with a different algorithm

HINT: The transformation logic is fundamentally wrong. Consider a simpler approach.

Figure 10: Prompt Templates - “complete_rewrite”
```

"major_restructure" (0.25 ≤ α < 0.50)

You are an AI assistant specialized in repairing Python code for ARC-AGI tasks.
Your goal is to analyze input-output grid pairs and repair a previously implemented transformation function.

STRICT OUTPUT REQUIREMENTS:

1. You MUST implement a function called transform.
2. The function MUST take one argument: the input grid as list[list[int]].
3. The function MUST return the transformed grid as a list[list[int]].
4. DO NOT return a raw numpy array or any other data type.

Task to solve:

{item['problem_description']}

Pattern Analysis from Examples:

{structural_hints}

Inferred Puzzle Properties:

{puzzle_properties_str}

Training Examples:

{training_examples_str}

Current Code to Repair (Champion):

{champion_code}

Current Accuracy: **{champion_accuracy:.1%}**

Feedback showing why it fails:

{final_feedback_payload}

Previously tried hypotheses (AVOID REPEATING THESE):

{previous_hypotheses}

Your task is to act as an expert code debugger. Follow these steps:

1. Analyze the Feedback: Inside <analysis> tags, summarize the key failures.
   Identify patterns in what's failing.
2. Formulate a New Hypothesis: Inside <hypothesis> tags,
   state a DIFFERENT hypothesis from previous attempts.
   This MUST fix the identified issues.
3. Plan the Code Changes: Inside <plan> tags, describe specific changes you'll make.
4. Write the Corrected Code: Provide the complete corrected implementation.

Your new hypothesis MUST be different from the current code's logic and from previously tried approaches.

HINT: Check transformation order, array indexing, and boundary conditions.

Figure 11: Prompt Templates - "major_restructure"

"targeted_fix" (0.50 ≤ α < 0.80)

You are an AI assistant specialized in repairing Python code for ARC-AGI tasks.
Your goal is to analyze input-output grid pairs and repair a previously implemented transformation function.

STRICT OUTPUT REQUIREMENTS:

1. You MUST implement a function called transform.
2. The function MUST take one argument: the input grid as list[list[int]].
3. The function MUST return the transformed grid as a list[list[int]].
4. DO NOT return a raw numpy array or any other data type.

Task to solve:

{item['problem_description']}

Pattern Analysis from Examples:

{structural_hints}

Inferred Puzzle Properties:

{puzzle_properties_str}

Training Examples:

{training_examples_str}

Current Code to Repair (Champion):

{champion_code}

Current Accuracy: **{champion_accuracy:.1%}**

Feedback showing why it fails:

{final_feedback_payload}

Previously tried hypotheses (AVOID REPEATING THESE):

{previous_hypotheses}

Your task is to act as an expert code debugger. Follow these steps:

1. Analyze the Feedback: Inside <analysis> tags, summarize the key failures. Identify patterns in what's failing.
2. Formulate a New Hypothesis: Inside <hypothesis> tags, state a DIFFERENT hypothesis from previous attempts. This MUST fix the identified issues.
3. Plan the Code Changes: Inside <plan> tags, describe specific changes you'll make.
4. Write the Corrected Code: Provide the complete corrected implementation.

Your new hypothesis MUST be different from the current code's logic and from previously tried approaches.

HINT: Focus on edge cases and special conditions that might be failing.

Figure 12: Prompt Templates - “targeted_fix”

"fine_tune" (0.80 ≤ α < 0.95)

You are an AI assistant specialized in repairing Python code for ARC-AGI tasks.
Your goal is to analyze input-output grid pairs and repair a previously implemented transformation function.

STRICT OUTPUT REQUIREMENTS:

1. You MUST implement a function called transform.
2. The function MUST take one argument: the input grid as list[list[int]].
3. The function MUST return the transformed grid as a list[list[int]].
4. DO NOT return a raw numpy array or any other data type.

Task to solve:

{item['problem_description']}

Pattern Analysis from Examples:

{structural_hints}

Inferred Puzzle Properties:

{puzzle_properties_str}

Training Examples:

{training_examples_str}

Current Code to Repair (Champion):

{champion_code}

Current Accuracy: **{champion_accuracy:.1%}**

Feedback showing why it fails:

{final_feedback_payload}

Previously tried hypotheses (AVOID REPEATING THESE):

{previous_hypotheses}

Your task is to act as an expert code debugger. Follow these steps:

1. Analyze the Feedback: Inside <analysis> tags, summarize the key failures. Identify patterns in what's failing.
2. Formulate a New Hypothesis: Inside <hypothesis> tags, state a DIFFERENT hypothesis from previous attempts. This MUST fix the identified issues.
3. Plan the Code Changes: Inside <plan> tags, describe specific changes you'll make.
4. Write the Corrected Code: Provide the complete corrected implementation.

Your new hypothesis MUST be different from the current code's logic and from previously tried approaches.

HINT: Almost there! Look for off-by-one errors or corner cases.

Figure 13: Prompt Templates - “fine_tune”

"precision_fix (0.95 ≤ α < 1.00)
(a) Standard precision fix

You are an AI assistant specialized in perfecting Python code for ARC-AGI tasks.

CRITICAL: The current solution has **{champion_accuracy:.1%}** accuracy - it's VERY CLOSE to perfect!
Only **{(1-champion_accuracy)*100:.1f}**% of pixels are wrong. Focus on the SPECIFIC failures.

Training Examples:

{training_examples_str}

Current Code (ALMOST PERFECT):

{champion_code}

Feedback showing why it fails (FOCUS ON THESE SPECIFIC FAILURES):

{final_feedback_payload}

Your task is to identify the pattern in the failures:

Provide the corrected Python code:
```
def transform(input_grid):
    # Your minimally modified solution
    pass
```

HINT: Almost there! Look for off-by-one errors or corner cases.

"precision_fix (0.95 ≤ α < 1.00)
(b) Stuck at High Accuracy Prompt (no improvement ≥ 2 iterations)

You are an AI assistant specialized in perfecting Python code for ARC-AGI tasks.

CRITICAL: The solution has **{champion_accuracy:.1%}** accuracy but has been stuck for **{no_improvement_count}**
iterations. This suggests you're missing a PATTERN, not just individual pixels.

Training Examples:

{training_examples_str}

Current Code (STUCK AT HIGH ACCURACY):

{champion_code}

Feedback showing why it fails:

{final_feedback_payload}

IMPORTANT: DO NOT add hardcoded fixes like:
- if grid[x, y] == value: new_grid[x, y] = value
- Special cases for specific coordinates
- Patches for individual pixels

Instead, identify the GENERAL RULE that would handle ALL these failures:

Then provide the corrected code:
```
def transform(input_grid):
    # Your solution with the general pattern
    pass
```

Figure 14: Prompt Templates - “precision_fix”

Level 2 : Detailed feedback

```
Content inserted into {final_feedback_payload}:

 --- Feedback on Training Example 1 ---
  The code resulted in a 'CONTENT_MISMATCH' with 67.3% pixel
accuracy.
  It failed to produce the required colors: [4, 7]
  It incorrectly introduced new colors: [2]
  There were 15 incorrect pixels.

  --- Feedback on Training Example 2 ---
  The code resulted in a 'CONTENT_MISMATCH' with 71.2% pixel
accuracy.
  There were 8 incorrect pixels.

  PARTIAL SUCCESS: Some patterns are being recognized correctly.
```

Figure 15: Level 2 “Detailed” Feedback

Level 3 : Pixel-Level feedback

```
Content inserted into {final_feedback_payload}:

  --- Feedback on Training Example 1 ---
  The code resulted in a 'CONTENT_MISMATCH' with 94.7% pixel
accuracy.
  There were 3 incorrect pixels:
    - At position [2, 5], expected color 4, but got 2
    - At position [3, 7], expected color 1, but got 0
    - At position [6, 1], expected color 3, but got 4

  CLOSE: Focus on edge cases and boundary conditions.
```

Figure 16: Level 3 “Pixel-level” Feedback

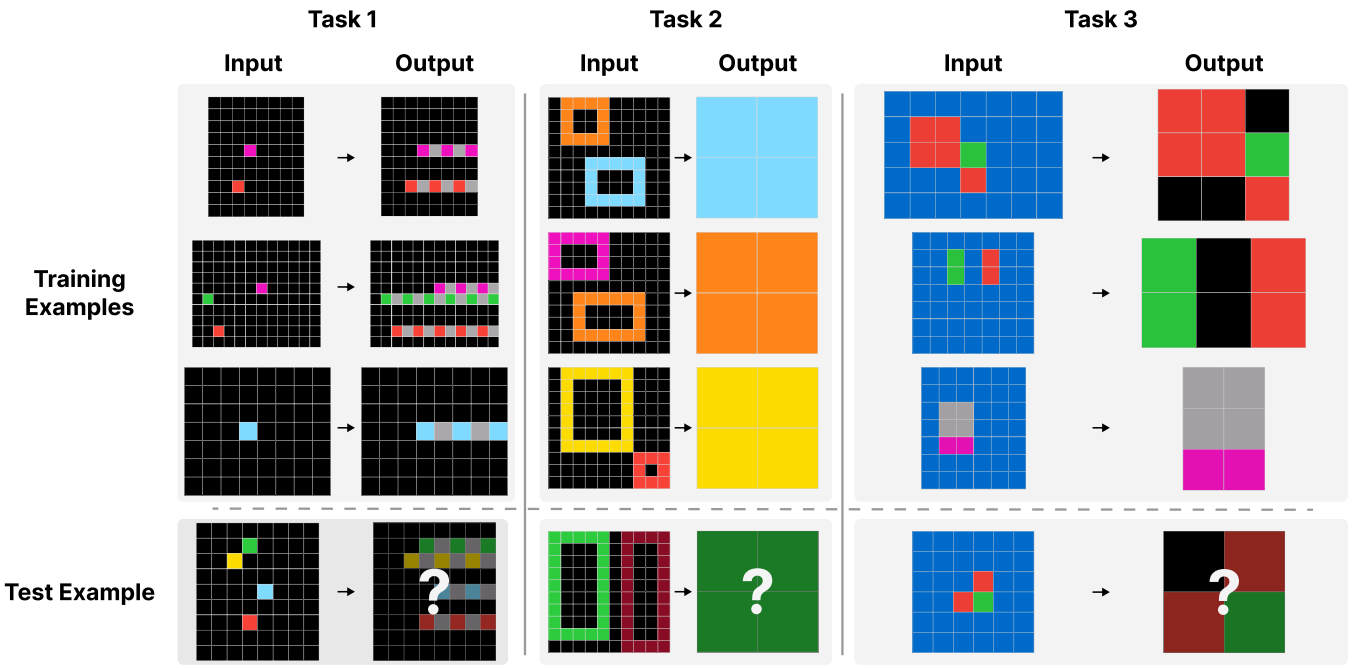


Figure 17: Examples ARC tasks. Each task is defined by input–output pairs where colored pixels represent objects or patterns. The goal is to infer the underlying transformation rule (e.g., recoloring, moving, duplicating, or resizing shapes) from the training examples and apply it to the test example (with '?' indicating the target to be predicted).

References

Balog, M.; Gaunt, A. L.; Brockschmidt, M.; Nowozin, S.; and Tarlow, D. 2017. DeepCoder: Learning to Write Programs. In *ICLR*. ICLR.

Bhattarai, M.; Cordova, M.; Santos, J.; and O'Malley, D. 2025. ARCS: Agentic Retrieval-Augmented Code Synthesis with Iterative Refinement. *arXiv:2504.20434*.

Bikov, K.; Bober-Irizar, M.; and Banerjee, S. 2025. MASR: Multi-Agent System with Reflection for the Abstraction and Reasoning Corpus. *OSF:388173109*.

Bober-Irizar, M.; and Banerjee, S. 2024. Neural Networks for Abstraction and Reasoning: Towards Broad Generalization in Machines. *arXiv:2402.03507*.

Butt, N.; Manczak, B.; Wiggers, A.; Rainone, C.; Zhang, D. W.; Defferrard, M.; and Cohen, T. 2024. CodeIt: Self-Improving Language Models with Prioritized Hindsight Replay. In *ICML*. ICML.

Chollet, F. 2019. On the Measure of Intelligence. *arXiv:1911.01547*.

Hui, B.; Yang, J.; Cui, Z.; Yang, J.; Liu, D.; Zhang, L.; Liu, T.; Zhang, J.; Yu, B.; Lu, K.; Dang, K.; Fan, Y.; Zhang, Y.; Yang, A.; Men, R.; Huang, F.; Zheng, B.; Miao, Y.; Quan, S.; Feng, Y.; Ren, X.; Ren, X.; Zhou, J.; and Lin, J. 2024. Qwen2.5-Coder Technical Report. *arXiv:2409.12186*.

Kuhn, M.; Johnson, K.; et al. 2013. *Applied Predictive Modeling*, volume 26. Springer.

Pourcel, J.; Colas, C.; and Oudeyer, P.-Y. 2025. Self-Improving Language Models for Evolutionary Program Synthesis: A Case Study on ARC-AGI. *arXiv:2507.14172*.

Siddiqi, N. 2012. *Credit Risk Scorecards: Developing and Implementing Intelligent Credit Scoring*, volume 3. John Wiley & Sons.

Zhong, L.; Wang, Z.; and Shang, J. 2024. Debug Like a Human: A Large Language Model Debugger via Verifying Runtime Execution Step by Step. In *ACL Findings*. ACL Findings.