Understanding Tool-Integrated Reasoning

Heng Lin
Tencent
Tsinghua University

Zhongwen Xu
Tencent
zhongwenxu@tencent.com

Abstract

We study why Tool-Integrated Reasoning (TIR) makes Large Language Models (LLMs) more capable. While LLMs integrated with tools like Python code interpreters show great promise, a principled theory explaining why this paradigm is effective has been missing. We provide the first formal proof that TIR fundamentally expands an LLM's capabilities, enabling previously impossible reasoning paths (Support Expansion) and making complex strategies practical within a finite token budget (Feasible Support). We conduct comprehensive experiments on challenging mathematical benchmarks, leveraging a Python interpreter as the external tool. Our results show that TIR models solve a class of problems that are fundamentally out of reach for pure-text models, even on tasks requiring deep abstract insight, not just calculation. We further identify the emergent cognitive patterns that illustrate how models learn to *think with tools*. To stably guide model behavior, we introduce Advantage Shaping Policy Optimization (ASPO), a novel algorithm that modifies the advantage directly, effectively encouraging desired tool-use behaviors without the training instability and performance loss of traditional reward shaping. Overall, our work provides the first principled explanation for TIR's success, shifting the focus from the mere fact *that* tools work to *why* and *how* they enable more powerful reasoning.

1 Introduction

Large language models (LLMs) have rapidly progressed from fluent generators to general-purpose problem solvers. Nevertheless, purely text-based reasoning often struggles with tasks that demand precise calculation, long-horizon search, or faithful verification. As a powerful and empirically successful paradigm, Tool-Integrated Reasoning (TIR) [4, 10] has emerged to address these limitations. Systems equipped with external tools have consistently and significantly outperformed their pure-text counterparts[11, 12, 17]. However, despite the widespread recognition of TIR's effectiveness, a principled account of the fundamental mechanisms—specifically *why* and *when* it helps—is still missing.

The central thesis of this work is that tool integration fundamentally breaks the capability ceiling of pure-text models. We provide the first formal proof that TIR enables a strict expansion of the model's empirical support by introducing deterministic, non-linguistic state transitions. This allows the model to generate correct trajectories that are otherwise impossible. Furthermore, we introduce the concept of token efficiency to argue that for many complex algorithms, natural language simulations are intractably verbose, making tools a practical necessity for unlocking a vastly larger set of feasible problem-solving strategies.

We validate these theoretical claims through comprehensive experiments on challenging mathematical benchmarks, showing that our TIR model decisively outperforms its pure-text counterpart. We also reveal that TIR's benefits are not confined to computationally-intensive problems but extend to those requiring significant abstract insight. Case studies of the model's behavior further illuminate *how* it leverages this expanded capability, revealing three emergent cognitive patterns of tool use.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: MATH-AI.

Finally, in exploring how to further optimize TIR models, we identify a practical algorithmic challenge: guiding model behavior, such as encouraging earlier tool use, via traditional reward shaping often leads to training instability in GRPO-like algorithms [15, 4]. To address this, we propose Advantage Shaping Policy Optimization (**ASPO**), a novel algorithm that circumvents the reward function and instead applies a stable, controllable bias directly to the advantage function. Our experiments show that ASPO successfully guides model behavior with early tool invocation and increased tool usages without compromising task performance or training stability.

2 Method

In this section, we formalize the argument that integrating an external computational tool, such as a code interpreter, fundamentally enhances a LLM's capabilities; similar principles apply for other tools and we have informal discussions in Appendix F. We then introduce ASPO, a novel and stable algorithm designed to guide the model's tool-usage behavior without compromising performance.

2.1 Support Expansion via Tool Integration

We begin by establishing that augmenting an LLM with a deterministic external tool enables it to generate trajectories that were previously impossible.

Our theory adopts the theoretical framework proposed by Wu et al. [16], which formalizes the limitations of standard on-policy reinforcement learning [3, 8, 14] on training LLMs. We briefly introduce the key concepts (a detailed review is provided in Appendix B). The **support** of a model with distribution p, $\text{supp}(p)$, is the set of all trajectories it can generate with non-zero probability. The **Support Preservation Theorem** [16] formalizes the "invisible leash" of Reinforcement Learning with Verifiable Rewards (RLVR), stating that the support of the RL-trained policy is a subset of the support of the base model. A more practical variant of support $\text{supp}(p)$ is the **empirical support**, $\text{supp}_\varepsilon(p)$, which only includes trajectories with a probability greater than a small threshold ε; in what follows, theory on TIR models will be made under this empirical-support view.

We compare a pure-text model (q_{text}) with a TIR model (p_{TIR}) that uses an identical underlying LLM but is augmented with a deterministic external tool (e.g., a Python interpreter). Now we present the first theorem and its proof sketch (a complete proof is provided in Appendix C):

Theorem 2.1 (Strict Expansion of Empirical Support via Tool Integration)**.** *There exists an $\varepsilon > 0$ and a family of problem instances such that*

$$\mathit{supp}_\varepsilon\big(q_{\mathit{text}}\big) \subset \mathit{supp}_\varepsilon\big(p_{\mathit{TIR}}\big).$$

Proof Sketch. **(Inclusion $\subseteq$)** is trivial, as the TIR model can simply choose not to use tool. **(Strictness $\subset$)** relies on a constructive proof using random oracle. The TIR model can deterministically solve the oracle problem in a single step. In contrast, the pure-text model must guess the high-entropy m-bit output, succeeding with a probability (2^{-m}) that becomes negligible for any practical threshold ε. □

Unlike pure-text models, which are constrained by Support Preservation Theorem, tool integration breaks the "invisible leash" and creates a strict expansion of the model's support.

2.2 Token Efficiency and Feasible Support under a Budget

While Theorem 2.1 shows tools unlock impossible trajectories, a deeper question is whether a pure-text model can achieve the same outcomes by *simulating* the computational process of tool use through natural language? The answer is no, due to a vast difference in **token efficiency**. We define the *token cost* of a trajectory, $\text{cost}(y)$, as the sum of all tokens consumed. For example, in any task involving iteration, a programmatic solution has a near-constant $O(1)$ token cost, whereas a natural language simulation's cost scales with the computation size. The tables in Appendix E illustrate this stark disparity for common algorithmic patterns. This motivates a practical, budget-aware analysis, based on the set of strategies a model can feasibly execute within a token budget B.

Definition 2.2 (Computational Equivalence Class)**.** Two trajectories are computationally equivalent if they solve the same problem using the same core algorithm. This relation partitions the space of all trajectories $\mathcal{Y}$ into equivalence classes, where each class $[y]$ represents an algorithmic "strategy".

Definition 2.3 (Feasible Support under Budget B)**.** An algorithmic strategy, represented by equivalence class $[y]$, is within the feasible support of a model M under token budget B, denoted $[y] \in \text{supp}_B(M)$, if and only if

$$\exists\, y' \in [y] \quad \text{s.t. } M(y'|x) > 0 \text{ and } \text{cost}(y') \leq B.$$

This definition captures a model's practical ability to realize a problem-solving strategy within operational constraints, which leads to our second theorem and proof (see details in Appendix D):

Theorem 2.4 (Strict Supremacy of Tool-Integrated Feasible Support)**.** *For any non-trivial algorithmic problem class and any token budget B, there exists a problem size n_B such that:*

$$\textit{supp}_B(q_{\textit{text}}) \subset \textit{supp}_B(p_{\textit{TIR}}).$$

Proof Sketch. **(Strictness $\subset$)** follows from the divergent scaling of the natural language. For any finite budget B, we can choose a problem size n_B large enough that the token cost of a natural language simulation exceeds B, while the $O(1)$ programmatic representation remains feasible. ☐

This theorem shows that tool integration is a necessity, as the verbosity of natural language makes many algorithmic strategies infeasible for pure-text models within a finite budget.

2.3 Algorithmic Improvement: Advantage Shaping Policy Optimization

To encourage a more dynamic tool-use style, we aimed to train the model to invoke tool *earlier*. However, a naive approach of adding an early-code bonus to reward proved highly unstable, because the group normalization of GRPO-like algorithm would catastrophically amplify the auxiliary bonus and distorts the learning objective in some cases (see details in Appendix G).

To circumvent this, we propose **Advantage Shaping Policy Optimization (ASPO)**. Instead of manipulating the reward, we directly modifies the final advantage value A_{correct} for correct and code-containing responses:

$$A_i = A_{\text{correct},i} + \text{clip}\left(\delta \cdot \frac{p_i - \text{mean}(\mathbf{p})}{\text{mean}(\mathbf{L})},\ \ -k \cdot A_{\text{correct},i},\ \ k \cdot A_{\text{correct},i}\right),$$

where $\mathbf{p}$ and $\mathbf{L}$ are the sets of first code invocation positions and total response lengths for all correct, code-containing responses within the group. δ is a negative coefficient to encourage early code invocation, and k is a clipping hyperparameter that bounds the magnitude of auxiliary advantage within a proportion of the basic advantage of correctness.

ASPO succeeds by directly applying a controlled bias to the advantage function. This ensures the primary correctness signal is never lost and the early-code incentive always remains a subordinate nudge, not the main objective (see detailed analysis in Appendix G). In essence, ASPO provides a robust and generalizable framework for shaping model behavior without sacrificing training stability or task performance, as shown in our experiments (Section 3.2).

3 Experiments

All experiments are based on the Qwen3-8B model [13], with a focus on challenging mathematical problem solving. We compare our proposed TIR model, which can execute code to assist in reasoning, against a pure-text RL baseline (see Figure 4 in Appendix H). Both models were trained using the DAPO algorithm [20]. Our primary evaluation benchmarks are AIME24, AIME25, and Omni-MATH-512, a challenging 512-problem subset of Omni-MATH [5]. Detailed setup is in Appendix H.

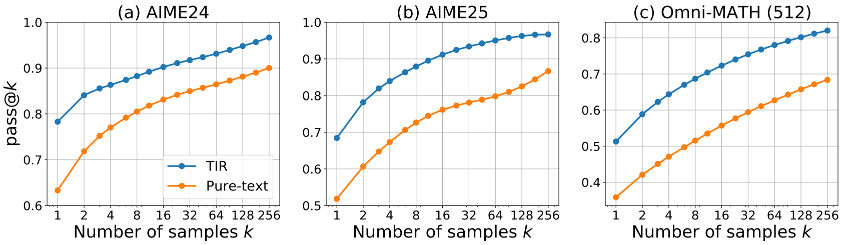

Figure 1: Pass@k curves for the TIR (RL trained) and pure-text models (Qwen3-8B) across three benchmarks: (a) AIME24, (b) AIME25, and (c) Omni-MATH-512. The detailed numerical data corresponding to this figure are provided in the Appendix I.

3.1 TIR Breaks the Capability Ceiling and is Universally Effective

To empirically test our theory, we use the pass@k metric [2], as it provides a robust measure of a model's underlying problem-solving potential. As shown in Figure 1, the TIR model's performance is unequivocally superior to the pure-text baseline across all benchmarks. Crucially, unlike prior findings for pure-text RL [21], the TIR model elevates the *entire* pass@k curve without the performance crossover at high k values. This provides strong empirical validation for our theory that TIR expands the model's support (Theorem 2.1 and 2.4). We also visualize the "flow of solvability" on the Omni-MATH-512 dataset in Figure 5 (Appendix J), which gives a microscopic analysis: on Omni-MATH-512, TIR achieves a **capability expansion** of 15.4%, with only 1.8% capability shrinkage, demonstrating a significant expansion of the model's support.

A critical question is whether this advantage is confined to purely computational problems. To investigate, we introduce a metric called "algorithmic friendliness," which scores problems from 1 (highly abstract) to 5 (directly algorithmic) based on a detailed rubric (see Appendix K). We used Gemini 2.5 Pro [6] to classify all Omni-MATH-512 problems, finding the dataset to be well-balanced and not skewed towards simple computational tasks (Figure 7(f) in Appendix L). We then plotted the pass@k curves for each algo friendliness group (Figure 7(a)-(e)), showing that TIR's benefits are universal. While the performance gap is largest on highly algorithmic problems (as expected), the TIR model maintains a significant and consistent advantage even on problems requiring the most abstract reasoning (friendliness scores 1.0-2.5).

This demonstrates that TIR's role extends beyond a simple calculator or a direct algorithm-implementer; the model is leveraging the code interpreter in more complex and sophisticated ways, learning to *think with tools*. The qualitative analysis on model outputs also support this, which reveals three emergent cognitive patterns of tool-use: **(1) Insight-to-computation transformation.** The model first engages in text-based reasoning to transform an abstract problem into a computationally tractable form. It then use the code to execute a algorithm (e.g., search, DP) on this newly formulated sub-problem. **(2) Exploration and verification via code.** This pattern is particularly prevalent in problems with low algo friendliness. For problems with unclear solution paths, the model uses the interpreter as an interactive sandbox. It formulates conjectures, writes short code snippets to test them, and iteratively refines its strategy based on the feedback, allowing it to discover insights through empirical experimentation. **(3) Offloading complex calculation.** In the most direct usage, the model delegates complex or tedious calculations to the interpreter. The first two patterns represent new algorithmic strategies that are infeasible for pure-text models due to prohibitive token costs. Such dynamic and flexible code invocation enables the TIR model to break the capability ceiling of its pure-text counterpart. Detailed analysis and examples of these patterns are provided in Appendix M.

3.2 Empirical Analysis of ASPO for Early Code Invocation

In this section, we empirically validate our ASPO algorithm, designed to encourage earlier code invocation. Figure 2 confirms that ASPO maintains stable training and final test accuracy, unlike the naive reward-shaping approach which quickly collapses. This stability ensures the primary goal of correctness is not sacrificed. Having established its safety, we confirm ASPO's effectiveness. As shown in Figure 3, ASPO dramatically shifts the model's behavior, reducing the average first code invocation position from 4,000 tokens to under 1,000. Concurrently, the model becomes a much

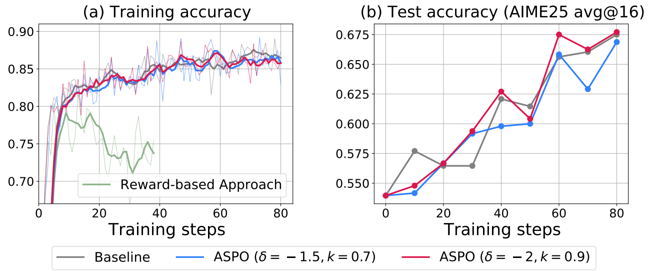

Figure 2: The (a) training and (b) testing accuracy of the baseline and ASPO algorithm, with two parameters setting: a *conservative* setting ($\delta = -2.0, k = 0.7$) and an *aggressive* setting ($\delta = -2.5, k = 0.9$).

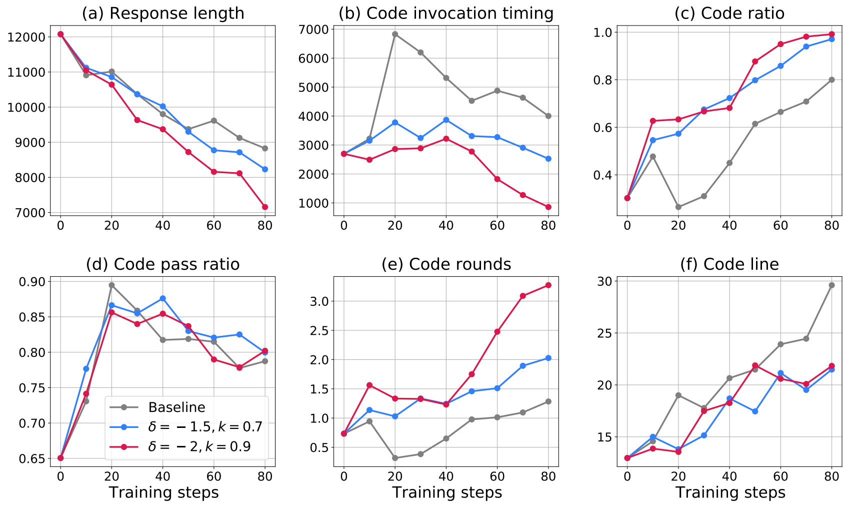

Figure 3: Evaluation results of the baseline and ASPO algorithm on AIME25. (a) Response length, (b) code invocation timing, (c) code ratio, (d) code pass ratio, (e) code rounds and (f) code lines.

more active tool user: the average number of code rounds per response more than doubles, from 1.3 to 3.3, and the code ratio approaches nearly 100%. This shows a clear transformation from a conservative, late-stage “calculator” usage pattern to an early, iterative, and exploratory “interactive partner” paradigm. Importantly, this behavioral shift is achieved without inducing reward hacking. We manually inspected a large number of samples and found no instances of the model inserting trivial or meaningless code early in its response merely to satisfy the incentive. Ultimately, ASPO demonstrates that it can precisely guide model’s behavior without compromising the stability or accuracy of its core learning objective.

4 Conclusions

In this work, we established a formal theoretical framework explaining *why* Tool-Integrated Reasoning (TIR) is effective. We proved that TIR fundamentally expands an LLM’s empirical and feasible support, breaking the “invisible leash” of pure-text models and making complex algorithmic strategies practically achievable. Our experiments empirically validated these claims, showing that TIR’s benefits are universal, extending even to problems requiring abstract reasoning. To address the challenge of guiding tool-use, we introduced ASPO, a stable and effective algorithm for shaping agent behavior. Ultimately, our findings advocate for a paradigm shift towards viewing LLMs as core reasoning engines that delegate tasks to specialized tools, a direction for which ASPO provides a robust control methodology.

References

[1] Fei Bai, Yingqian Min, Beichen Zhang, Zhipeng Chen, Wayne Xin Zhao, Lei Fang, Zheng Liu, Zhongyuan Wang, and Ji-Rong Wen. Towards effective code-integrated reasoning. *arXiv preprint arXiv:2505.24480*, 2025.

[2] 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.

[3] Team DeepSeek. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645:633–638, 2025.

[4] Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. ReTool: Reinforcement learning for strategic tool use in LLMs. *arXiv preprint arXiv:2504.11536*, 2025.

[5] Bofei Gao, Feifan Song, Zhe Yang, Zefan Cai, Yibo Miao, Qingxiu Dong, Lei Li, Chenghao Ma, Liang Chen, Runxin Xu, et al. Omni-math: A universal olympiad level mathematic benchmark for large language models. *arXiv preprint arXiv:2410.07985*, 2024.

[6] Team Gemini. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

[7] Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-R1: Training LLMs to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.

[8] Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, et al. Tülu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*, 2024.

[9] Kuan Li, Zhongwang Zhang, Huifeng Yin, Liwen Zhang, Litu Ou, Jialong Wu, Wenbiao Yin, Baixuan Li, Zhengwei Tao, Xinyu Wang, et al. WebSailor: Navigating super-human reasoning for web agent. *arXiv preprint arXiv:2507.02592*, 2025.

[10] Xuefeng Li, Haoyang Zou, and Pengfei Liu. ToRL: Scaling tool-integrated RL. *arXiv preprint arXiv:2503.23383*, 2025.

[11] OpenAI. Introducing GPT-5. Blog post, Aug 2025. URL `https://openai.com/index/gpt-5/`. Accessed on August 22, 2025.

[12] OpenAI. Introducing o3 and o4-mini. Blog post, April 2025. URL `https://openai.com/index/introducing-o3-and-o4-mini/`. Accessed on August 22, 2025.

[13] Team Qwen. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[14] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[15] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[16] Fang Wu, Weihao Xuan, Ximing Lu, Zaid Harchaoui, and Yejin Choi. The invisible leash: Why RLVR may not escape its origin. *arXiv preprint arXiv:2507.14843*, 2025.

[17] xAI. Grok 4. Blog post, Jul 2025. URL `https://x.ai/blog/grok-4`. Accessed on August 22, 2025.

[18] Zhongwen Xu, Xianliang Wang, Siyi Li, Tao Yu, Liang Wang, Qiang Fu, and Wei Yang. Agents play thousands of 3D video games. *arXiv preprint arXiv:2503.13356*, 2025.

[19] Zhenghai Xue, Longtao Zheng, Qian Liu, Yingru Li, Zejun Ma, and Bo An. SimpleTIR: End-to-end reinforcement learning for multi-turn tool-integrated reasoning. *arXiv preprint arXiv:2509.02479*, 2025.

[20] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, et al. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

[21] Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in LLMs beyond the base model? *arXiv preprint arXiv:2504.13837*, 2025.

A Related Work

A significant body of work focuses on developing RL frameworks for strategic tool use. Feng et al. [4] propose ReTool, an RL-based framework that demonstrates high data efficiency for learning tool use. Similarly, Li et al. [10] introduce ToRL, a method designed to address the challenges of scaling tool-integrated RL to more complex and demanding scenarios. Bai et al. [1] document methods for effective code-integrated reasoning. This paradigm shares the goal of augmenting LLM reasoning with external Python execution. Focusing on training from base models, Xue et al. [19] present SimpleTIR, an end-to-end framework for multi-turn TIR that enables stable training from scratch, a process they refer to as the “Zero” setting.

While these methods show empirical success, other research investigates the theoretical limitations of RL on LLM reasoning. Yue et al. [21] empirically find that RL does not incentivize novel reasoning capacity. Providing a theoretical framework to explain such findings, Wu et al. [16] propose the “invisible leash” theory, suggesting that models may struggle to discover reasoning paths outside their original knowledge distribution.

Beyond programmatic tools like Python interpreters, another line of work integrates search engines to equip LLMs with up-to-date knowledge via RL. Jin et al. [7] propose Search-R1, where LLMs interleave reasoning with real-time queries, trained with outcome-based rewards and stabilized by masking retrieved tokens, achieving strong multi-turn QA performance. To tackle uncertainty in complex web tasks, Li et al. [9] introduce WebSailor, a post-training method that narrows the gap with proprietary agents.

B The Theoretical Background

To ground our proof, we adopt the theoretical framework proposed by Wu et al. [16], which formalizes the limitations of standard on-policy reinforcement learning [3, 8, 14] on training LLMs.

Definition B.1 (Support of a Model (adapted from [16]))**.** Let $\mathcal{Y}$ be the space of all possible generative trajectories. The support of a model with distribution $p(y|x)$ is the set of all trajectories that can be generated with a non-zero probability for a given prompt x:

$$\text{supp}(p) := \{y \in \mathcal{Y} \mid p(y|x) > 0\}$$

Definition B.2 (Empirical Support (from Wu et al. [16]))**.** For a threshold $\varepsilon > 0$, define the empirical support of p as

$$\text{supp}_\varepsilon(p) := \{y \in \mathcal{Y} \mid p(y|x) \geq \varepsilon\}.$$

This definition is central to understanding a model’s intrinsic capabilities. The following theorem from Wu et al. [16] establishes a key constraint for models trained with Reinforcement Learning from Verifiable Rewards (RLVR) [8, 3]:

Theorem B.3 (Support Preservation under RLVR (from Wu et al. [16]))**.** *Let $\pi_\theta(y|x)$ be an RLVR-trained policy distribution initialized from a base model with distribution $q(y|x)$. For any prompt x, the support of the trained policy is a subset of the support of the base model:*

$$\textit{supp}(\pi_\theta) \subseteq \textit{supp}(q)$$

This implies that if $q(y^*|x) = 0$ for a correct trajectory y^*, then RLVR can never discover y^*.

Theorem B.3 formalizes the “invisible leash”: RLVR can only re-weight probabilities within the model’s pre-existing support. We next show a strictly stronger, practical statement under an empirical-support view.

C The Detailed Proof of Support Expansion

We consider two types of LLMs in this work. A pure-text model is a standard language model with distribution q_{text} that generates tokens exclusively from its vocabulary $\mathcal{V}$. We compare this to a tool-integrated model, a system $(M, \mathcal{O})$ with distribution p_{TIR}, which pairs a language model M with a deterministic external oracle $\mathcal{O}$ (e.g., a Python interpreter). The generative process for this model includes not only probabilistic token generation from $\mathcal{V}$ but also deterministic tool-use transitions. In such a transition, the model M emits a tool call y_{call}, the oracle executes it, and the resulting output $y_{\text{out}} = \mathcal{O}(y_{\text{call}})$ is deterministically returned as the next state.

Now we present the detailed proof of Theorem 2.1.

Proof. The proof proceeds in two parts. First, we establish the subset relationship ($\subseteq$), and second, we prove the relationship is strict ($\neq$) by demonstrating the existence of trajectories accessible only to the tool-integrated model.

Part 1: Proving supp(q_{text}) $\subseteq$ supp(p_{TIR})

Let y be an arbitrary trajectory in the support of the pure-text model, such that $q_{\text{text}}(y|x) > 0$. The trajectory y consists exclusively of tokens from the vocabulary $\mathcal{V}$. The tool-integrated model p_{TIR} can generate this same trajectory by adopting a policy of never invoking the external oracle $\mathcal{O}$. Since its generative capabilities subsume those of q_{text}, it can assign a non-zero probability to the trajectory y. Thus, for any $y \in \text{supp}(q_{\text{text}})$, it follows that $y \in \text{supp}(p_{\text{TIR}})$, establishing that $\text{supp}(q_{\text{text}}) \subseteq \text{supp}(p_{\text{TIR}})$.

Part 2: Proving Strictness

To prove strictness, we use a constructive approach based on a standard cryptographic primitive: a *random oracle*. Let us consider a problem instance where the solution requires computing $y_{\text{out}} = H(x)$, where H is a random oracle. A random oracle is a theoretical black box that, for any new input query, returns an output chosen uniformly at random from its output space (e.g., $\{0, 1\}^m$), but deterministically returns the same output for repeated queries of the same input. This construction is theoretically convenient and serves as an idealization of practical cryptographic hash functions (e.g., SHA-256). For a model without access to the oracle, its only strategy to find y_{out} is to guess it. The probability of correctly guessing a specific m-bit string is 2^{-m}.

Now, consider a trajectory $y^* = (y^*_{\text{prefix}}, y_{\text{out}}, y^*_{\text{suffix}})$ that involves computing $H(x)$. We assume the underlying language model for both p_{TIR} and q_{text} is identical. The tool-integrated model p_{TIR} can invoke the oracle to obtain y_{out} deterministically. In contrast, the pure-text model, q_{text}, must guess y_{out} from an output space of size 2^m, succeeding with a probability of only 2^{-m}. Thus, the total probabilities of producing y^* are directly related:

$$q_{\text{text}}(y^*|x) = p_{\text{TIR}}(y^*|x) \cdot 2^{-m}.$$

For any non-negligible probability $p_{\text{TIR}}(y^*|x)$ and a sufficiently large m, the corresponding $q_{\text{text}}(y^*|x)$ becomes arbitrarily small. We can therefore always choose an ε such that $q_{\text{text}}(y^*|x) < \varepsilon \leq p_{\text{TIR}}(y^*|x)$. So we find that $y^* \notin \text{supp}_\varepsilon(q_{\text{text}})$ while $y^* \in \text{supp}_\varepsilon(p_{\text{TIR}})$. This establishes strictness. $\square$

D The Detailed Proof of Feasible Support Supremacy

Here we present the detailed proof of Theorem 2.4.

Proof. The proof requires showing both inclusion ($\subseteq$) and strictness ($\neq$).

Inclusion ($\subseteq$): Any algorithmic strategy that is feasibly executable by a pure-text model within budget B is, by definition, also executable by a tool-integrated model that simply abstains from using its tool.

Strictness ($\neq$): We must show there exists an algorithmic class $[y_A]$ in $\text{supp}_B(p_{\text{TIR}})$ but not in $\text{supp}_B(q_{\text{text}})$. This follows directly from the divergent scaling properties of natural language versus programmatic representations, as illustrated in Tables 1-4. For any algorithm whose pure-text simulation cost scales with problem size n (e.g., $\Omega(n)$, $\Omega(V + E)$), we can choose a size n_B such that the cost exceeds any finite budget B. The programmatic representation, costing $O(1)$, remains within budget. Thus, for a sufficiently large problem size, the corresponding algorithmic classes are in the feasible support of p_{TIR} but not q_{text}, proving strict inclusion. $\square$

E Examples on Token Efficiency

Table 1: Contrasting Token Efficiency for an Iterative Task ($N \to \infty$)

Programmatic Approach (Python)	**Natural Language Reasoning**
A symbolic, abstract representation of the computation. The token cost is constant and independent of N.	A concrete, step-by-step enumeration of the computation. The token cost scales with the magnitude of N.

`# N can be 10,000,000 or more`
`for i in range(N):`
`  # Perform some check`
`  check(i)`
	"Okay, to solve this, I must check every number.
	First, for n=1, I perform the check...
	Next, for n=2, I perform the check...
	Next, for n=3, I perform the check...
	...
	(This enumeration continues for millions of steps)
	...
	Finally, for n=10,000,000, I perform the check..."

Token Cost: A few dozen tokens. Scales as $O(1)$. This is highly efficient and scalable.	**Token Cost:** Proportional to N. Scales as $\Omega(N)$. This is inefficient and becomes intractable for large N, quickly exceeding any feasible context window.

Table 2: Contrasting Token Efficiency for Solving Large Linear Systems

Programmatic Approach (Python)	**Natural Language Reasoning**
A single call to a highly optimized numerical library solves $Ax = b$. The token cost is constant, independent of the matrix dimension n.	A detailed explanation of Gaussian elimination, requiring a description of each row operation. The token cost scales with the matrix size.

`import numpy as np`
`# A is a large n x n matrix,`
`# e.g., n=1000`
`x = np.linalg.solve(A, b)`
	"To solve the system, we perform Gaussian elimination. First, to eliminate the first variable from the second row, we subtract $A_{2,1}/A_{1,1}$ times the first row from the second row. We must do this for all $n-1$ rows below the first. Next, we use the new second row to eliminate the second variable from the rows below it... (This narration continues for $O(n^2)$ elements and $O(n^3)$ operations)."

Token Cost: A few tokens. Scales as $O(1)$. Enables solving massive systems within a tiny token budget.	**Token Cost:** Proportional to the number of elements in the matrix to sketch. Scales as $\Omega(n^2)$. A full narration would scale as $\Omega(n^3)$.

Table 3: Contrasting Token Efficiency for a Dynamic Programming Task (Fibonacci Sequence)

Programmatic Approach (Python)	**Natural Language Reasoning**
A compact representation of the recurrence relation, with a token cost independent of the input integer N.	A verbose, step-by-step calculation of every subproblem's solution, with a token cost that grows with N.
	"To get fib(5), I need fib(4) and fib(3). Fib(2) is fib(1)+fib(0) = 1+0 = 1. Fib(3) is fib(2)+fib(1) = 1+1 = 2. Fib(4) is fib(3)+fib(2) = 3+1 = 4. So, fib(5) is fib(4)+fib(3) = 4+2 = 6... Wait, let me recheck. fib(4) is 3+2=5. No, fib(4) is 2+1=3. Okay, so fib(5) is 3+2=5."
Token Cost: $O(1)$	**Token Cost:** $\Omega(N)$

```python
memo = {0: 0, 1: 1}
def fib(n):
  if n in memo: return memo[n]
  memo[n] = fib(n-1) + fib(n-2)
  return memo[n]
```

Table 4: Contrasting Token Efficiency for Search Algorithms

Programmatic Approach (Python)	**Natural Language Reasoning**
An abstract procedure for state-space traversal, using data structures like a queue and a set.	A full, step-by-step narration of the entire exploration process, including every node visited and every state change of the queue.
	"I start at node 'A'. Queue is ['A'], visited is 'A'. I pop 'A'. Its neighbors are 'B', 'C'. Queue is now ['B', 'C'], visited is 'A','B','C'. I pop 'B'. Its neighbor is 'D'. Queue is now ['C', 'D'], visited is 'A','B','C','D'. I pop 'C'..." (and so on)
Token Cost: Constant cost for the algorithm's definition. Scales as $O(1)$.	**Token Cost:** Proportional to the number of vertices and edges, $V + E$. Scales as $\Omega(V + E)$.

```python
from collections import deque

def bfs(graph, start_node):
  queue = deque([start_node])
  visited = {start_node}
  while queue:
    node = queue.popleft()
    # Process node
    for neighbor in graph[node
    ]:
      if neighbor not in
    visited:
          visited.add(neighbor)
          queue.append(neighbor)

```

F Extensions to Other Tools and Interactions with Environments

Our arguments in Sections 2.1 and 2.2 extend beyond Python to a broad family of external tools and interactive settings. At a high level, any interface that (i) affords *state transitions* not expressible by next-token sampling alone and/or (ii) delivers *high information per token of I/O* will both expand support (Section 2.1) and strictly enlarge feasible support under a token budget (Section 2.2).

Search and Retrieval Agents. Consider web search, retrieval APIs, or domain databases (e.g., scholarly indices, code search). Let an external retriever implement a (possibly stochastic) mapping $\mathcal{R} : (q, s) \mapsto r$, where q is a query issued by the LLM and s is the (latent) world/index state at the time of the call. Even when $\mathcal{R}$ is not perfectly deterministic, the *trajectory* that includes the returned snippet r is unreachable for a pure-text model unless it *guesses* the salient facts in r token-by-token. This mirrors the random-oracle argument in Theorem 2.4: as the entropy of r conditioned on (q, x) grows, the probability that a pure-text model reproduces r by chance decays exponentially, while a tool-augmented model obtains r via a single call. Hence support expands, and under any fixed budget B the feasible set also strictly expands once the text-only paraphrase of r would exceed B.

Checkers, Verifiers, and Program Runners. Beyond "heavy" computation, many tools act as *verifiers*: unit tests, symbolic algebra checkers, SAT/SMT solvers, theorem provers, type checkers, or even a Python REPL used only to validate a candidate answer. Such tools add *deterministic pruning* transitions to the trajectory graph: incorrect branches are cut immediately with $O(1)$ tokens. This reduces the exploration burden under RLVR-style training and enlarges the set of practically reachable strategies under a budget.

Stateful External Memory. Tools can expose memory larger and more persistent than the model's context: key–value caches, external scratchpads, vector stores, or file systems. Each call updates an external state $m_{t+1} = U(m_t, a_t)$ and reads views $v_t = V(m_t)$ at $O(1)$ token cost. Strategies that require memory $|m| \gg B$ are impossible to realize faithfully in pure text (which must inline m), but become feasible when memory lives outside the context window.

Proposition F.1 (Informal; External State as Unbounded Scratchpad)**.** *Suppose an algorithm requires $\Omega(n)$ writable memory cells for problem size n. If a tool exposes these cells with per-step I/O $O(1)$, then for sufficiently large n, the algorithm's equivalence class lies in $\mathrm{supp}_B(p_{\mathrm{TIR}})$ but not in $\mathrm{supp}_B(q_{\mathrm{text}})$ for any fixed B.*

Embodied and Interactive Environments. When the LLM acts in an MDP or game environment [18], the environment transition $s_{t+1} = E(s_t, a_t)$ is itself an *external oracle*. Our earlier support-expansion argument applies verbatim: trajectories that include specific environment observations or states are unreachable by text-only generation unless they are guessed token-by-token. Token-efficiency arguments also lift: environment interactions can realize long-horizon plans with *summarized* textual traces, whereas a pure-text simulation would require enumerating each counterfactual step.

Noisy or Non-Deterministic Tools. Stochastic returns (e.g., fluctuating search rankings) do not invalidate support expansion. What matters is the existence of *some* positive-probability outputs with substantial conditional entropy that are infeasible to reproduce via text within budget. In other words, determinism is a convenience, not a necessity, for our conclusions.

Composing Multiple Tools. Real agents chain retrieval, computation, verification, and environment actions. Composition behaves monotonically:

Proposition F.2 (Informal; Monotone Closure under Composition)**.** *Let $\mathcal{T}_1, \ldots, \mathcal{T}_k$ be tools with per-call costs that sum to at most B. If each $\mathcal{T}_i$ individually yields a strict feasible-support gain for some subproblem family at size n_i, then there exist composite tasks for which the sequential (or branched) use of $\{\mathcal{T}_i\}$ yields a strict feasible-support gain over any pure-text policy at the same total budget.*

Takeaway. "Python" is merely one instantiation of a broader principle, our extensions unify code execution, search, verification, memory, and embodied interaction under the same analytical lens.

G Detailed Analysis of ASPO

The TIR models often default to a conservative strategy: completing the majority of their abstract reasoning via text before invoking the code interpreter for the final-step calculation or verification. This overlooks a potentially more powerful paradigm where the interpreter is used as an exploratory tool throughout the reasoning process. We hypothesize that encouraging the model to invoke code *earlier* could foster a more dynamic, flexible, and hypothesis-driven reasoning style, potentially unlocking novel problem-solving strategies.

To encourage the earlier code invocation, our initial and most direct approach was to introduce an *early-code reward* directly into the reward function. For each response i that is both correct and code-containing in a group of samples, we added a reward term r'_i that penalizes later code invocation:

$$R_i = 1 + r'_i \quad \text{where} \quad r'_i = \delta \cdot \text{clip}\left(\frac{p_i - \text{mean}(\mathbf{p})}{\text{std}(\mathbf{p})}, -c, c\right).$$

where $\mathbf{p}$ is the set of first code invocation positions for all correct, code-containing responses within the group. Furthermore, δ is a negative coefficient to encourage early code invocation, and c is a clipping hyperparameter. However, this seemingly innocuous modification proved to be highly destabilizing during training (see experimental details in Section 3.2 and Figure 2 (a)). In algorithms like GRPO that rely on group normalized advantage, this design has a critical flaw. In the common scenario where all samples in a group are correct, the primary reward signal (the constant '1') is entirely eliminated by the normalization. The advantage calculation then becomes:

$$A_i = \frac{R_i - \text{mean}(\mathbf{R})}{\text{std}(\mathbf{R})} = \frac{(1 + r'_i) - (1 + \text{mean}(\mathbf{r}'))}{\text{std}(\mathbf{r}')} = \frac{r'_i - \text{mean}(\mathbf{r}')}{\text{std}(\mathbf{r}')}.$$

This leads to a catastrophic outcome: (1) the primary signal about answer correctness disappears, (2) the auxiliary signal r'_i is amplified to the same magnitude as the original primary signal, and (3) due to the nature of standardization, approximately half of these correct responses receive a negative advantage and are thus heavily penalized, solely because their code invocation is later than the group's average.

To circumvent the distorting effects of reward normalization, we developed a more robust method that we term Advantage Shaping Policy Optimization (**ASPO**). Instead of manipulating the reward, we directly modify the final advantage value after the standard correctness-based advantage A_{correct} has been calculated. For any response i that is both correct and contains code, we compute the new advantage A_i as follows:

$$A_i = A_{\text{correct},i} + \text{clip}\left(\delta \cdot \frac{p_i - \text{mean}(\mathbf{p})}{\text{mean}(\mathbf{L})},\ -k \cdot A_{\text{correct},i},\ k \cdot A_{\text{correct},i}\right),$$

where $\mathbf{p}$ and $\mathbf{L}$ are the sets of first code invocation positions and total response lengths for all correct, code-containing responses within the group. Furthermore, δ is a negative coefficient to encourage early code invocation, and k is a clipping hyperparameter that bounds the magnitude of auxiliary advantage within a proportion of the basic advantage of correctness.

This formulation has several key merits, primarily by circumventing the uncontrollable effects of advantage normalization inherent to reward-based modifications. First, it addresses a critical flaw in the reward-based approach: the inability to guarantee a positive advantage for all correct answers. After adding the auxiliary reward, a correct response's total reward could fall below the group average, leading to a negative GRPO normalized advantage, which effectively punishes a correct solution. Second, the GRPO normalization process itself introduces uncontrollable volatility: the std($\mathbf{R}$) in the denominator unpredictably scales the auxiliary signal, making its influence inconsistent across different groups.

Our ASPO algorithm resolves both issues. By applying a clipped bias directly to A_{correct}, we ensure the final advantage remains positive and that the early-code incentive is always a subordinate nudge, never overwhelming the primary objective of correctness. Furthermore, this approach bypasses the volatile scaling effect of std($\mathbf{R}$) entirely. Finally, the choice to normalize the code invocation position by the mean response length mean($\mathbf{L}$) rather than the standard deviation of positions std($\mathbf{p}$) is deliberate. The latter is unstable: when invocation positions in a group are tightly clustered, a small std($\mathbf{p}$) would excessively amplify the signal, whereas a more stable denominator like mean($\mathbf{L}$) is consistent and meaningful. This method allows us to stably and effectively encourage early code invocation, the empirical results of which are detailed in Section 3.2.

In essence, ASPO provides a general and robust framework for guiding a model's behavior towards desired styles or properties without compromising the primary learning objective (e.g., accuracy). By directly manipulating the advantage values, ASPO avoids the instabilities that can arise from altering the reward function, particularly in GRPO-like algorithms that rely on reward normalization. This method ensures that the incentive for the desired behavior (in this case, earlier code invocation) acts as a stable adjustment. The core principles of ASPO could be *readily adapted* to encourage other desirable behaviors in a variety of scenarios, offering a reliable approach to shape model conduct while preserving training stability and overall task performance.

H Experimental Setup

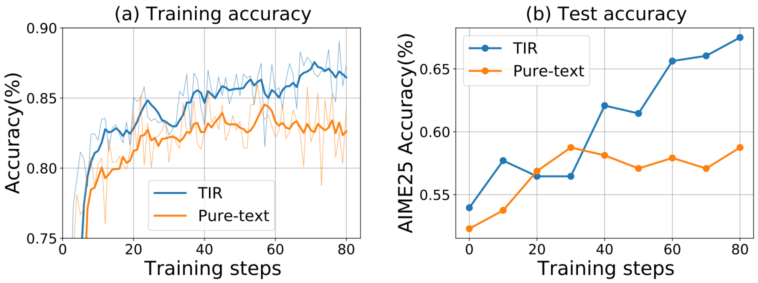

Figure 4: The (a) training and (b) testing accuracy of the TIR and pure-text RL on Qwen3-8B model. The AIME25 accuracy (b) is the average of 16 responses.

Model and Datasets. All experiments are based on the Qwen3-8B model [13]. For our training data, we randomly sample 10,000 English problems from the DAPO dataset [20] due to limited computational resources. Since our aim is to fundamentally understand the mechanisms of TIR *rather than* to improve absolute accuracy of benchmarks, this dataset is sufficient for our purpose, in contrast to the extensive training datasets used in other literature [20, 4]. Our primary evaluation benchmarks are AIME24, AIME25, and a challenging subset of the Omni-MATH dataset [5]. For the latter, due to the large size of the dataset, we curated the 512 *most difficult* problems that are amenable to reliable, rule-based evaluation, which we denote as Omni-MATH-512.

Training Protocol. We train two main models for comparison: our proposed TIR model, which can execute code to assist in its reasoning process, and a pure-text RL model as a baseline (as shown in Figure 4). Both models are trained for 3 epochs using the DAPO algorithm [20], a variant of GRPO [3]. During training, we use a rollout batch size of 96 problems, with 8 responses sampled per problem, a maximum response length of 16,384 tokens, and a sampling temperature of 1.0 to encourage exploration.

Evaluation Protocol. For evaluations, we set the sampling temperature to 0.6 and maximum response length to 16,384 tokens unless otherwise specified.

I Pass@k Data

Table 5 shows the detailed pass@k results for the TIR and pure-text models across the three benchmarks, evaluated with the max sample size of 256.

Table 5: Pass@k results for the TIR model and the pure text model

k	**AIME24**		**AIME25**		**Omni-MATH-512**	
	TIR	**Pure Text**	**TIR**	**Pure Text**	**TIR**	**Pure Text**
1	0.7829	0.6331	0.6841	0.5184	0.5128	0.3585
2	0.8408	0.7184	0.7818	0.6065	0.5885	0.4208
4	0.8632	0.7703	0.8395	0.6730	0.6437	0.4707
8	0.8825	0.8050	0.8792	0.7262	0.6869	0.5153
16	0.9024	0.8312	0.9117	0.7613	0.7232	0.5570
32	0.9173	0.8496	0.9339	0.7810	0.7545	0.5942
64	0.9312	0.8645	0.9503	0.7979	0.7802	0.6271
128	0.9480	0.8813	0.9625	0.8250	0.8018	0.6575
256	0.9667	0.9000	0.9667	0.8667	0.8203	0.6836

J Capability Expansion and Shrinkage

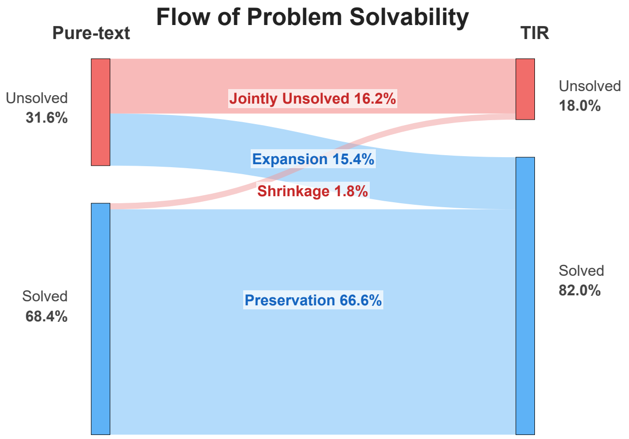

Figure 5: The flow of problem solvability on Omni-MATH-512 when transitioning from the pure-text model to the TIR model, evaluated at $k = 256$. A detailed version is provided in Figure 6.

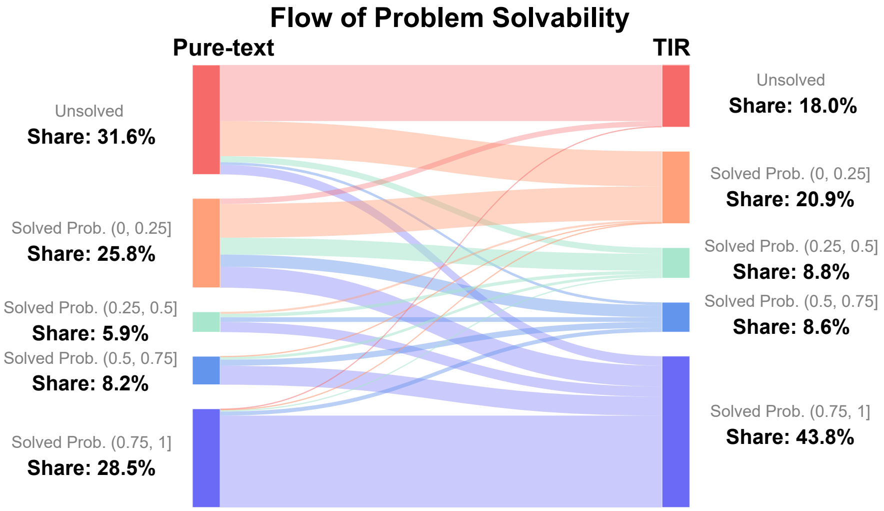

Figure 6: The detailed flow of problem solvability on Omni-MATH-512 when transitioning from the pure-text model to the TIR model. The solved probability of each problem is evaluated at $k = 256$.

K Rubric for Algorithmic Friendliness

Table 6 shows the rubric we use for Gemini Pro APIs [6] to classify the math problems.

Table 6: Rubric for assessing the "algorithmic friendliness" of problems.

Score	Level	Description	Required Insight
5	**Very High** (Direct Application)	The problem is a textbook example for a standard algorithm (e.g., backtracking). The problem statement itself almost serves as the specification. **Almost no mathematical insight is needed**.	None beyond basic arithmetic.
4	**High** (Minor Insight)	An algorithm provides a clear advantage, but requires a **standard, well-known mathematical identity** or **simple transformation** to be applied. The mathematical hurdle is low.	Recalling and applying a common formula or theorem.
3	**Medium** (Significant Insight)	A computational solution is effective, but only after applying a **significant mathematical insight** or performing **complex problem modeling**. The difficulty is substantial.	A creative, problem-specific trick or a complex modeling effort.
2	**Low** (Impractical Algorithm)	An algorithm is theoretically possible but highly impractical (enormous search space, precision issues). The algorithmic optimizations are **equivalent in difficulty to the mathematical solution**.	Insights needed are essentially the mathematical solution itself.
1	**Very Low** (Non-computational)	The problem is fundamentally abstract and cannot be solved by computation (e.g., requires a formal proof, deals with uncountable sets).	N/A.

L Pass@k Curves for Problems Grouped by Algo Friendliness

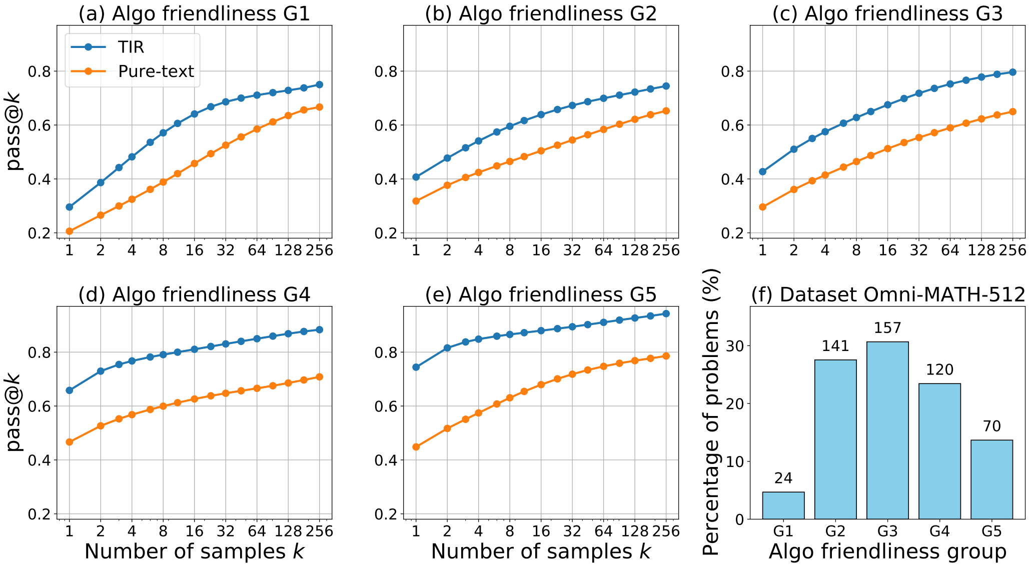

Figure 7: (a)-(e) Pass@k curves for the TIR and pure-text models, grouped by problem algo friendliness. (f) The distribution of algo friendliness scores across the Omni-MATH-512 dataset. The problems are categorized into five groups based on their algo friendliness scores: 1.0–1.5 (G1), 2.0–2.5 (G2), 3.0–3.5 (G3), 4.0–4.5 (G4), and 5.0 (G5).

M Emergent Cognitive Patterns of Tool Integration

The quantitative results of the previous sections demonstrate *that* TIR is universally effective, but they do not fully explain *how*. If the model's advantage is not limited to algorithmically amenable problems, how exactly is it leveraging the code interpreter to solve problems requiring abstract insight? Through qualitative analysis of model outputs, we have identified three distinct and recurring patterns of code utilization that answer this question.

Pattern 1: Insight-to-computation transformation. In this primary pattern, the model's first step is not to code, but to *reason*. It engages in non-trivial, text-based analysis to deconstruct a complex problem, applying mathematical insights to transform it into a state that is amenable to a programmatic solution. The code interpreter is then invoked to execute a genuine algorithm (such as search, enumeration, or DP) that efficiently solves this newly formulated sub-problem under a limited computational resource. Unlike straightforward sequential calculations one might perform with a calculator, these algorithms often possess complex control flows (e.g., loops, recursion) that are challenging for a language model to emulate or follow step-by-step.

For instance, as shown in Table 7, the model first uses mathematical reasoning to derive a transcendental equation from the abstract geometric problem. It then employs code to iterate the entire parameter space of (m, n) pairs, using the Intermediate Value Theorem as a numerical method to efficiently detect whether a solution exists for each pair.

Table 7: A case of insight-to-computation transformation.

[Question]
Let $a \neq b$ be positive real numbers and m, n be positive integers. An $m + n$-gon P has the property that m sides have length a and n sides have length b. Further suppose that P can be inscribed in a circle of radius $a + b$. Compute the number of ordered pairs (m, n), with $m, n \leq 100$, for which such a polygon P exists for some distinct values of a and b.

[Response]
Okay, let's try to tackle this problem step by step.
...
We need to find the number of ordered pairs (m, n) where $m, n \leq 100$ such that this equation has solutions for a and b with $a \neq b$. ... So, substituting into the equation:

$$m \cdot \arcsin(x/2) + n \cdot \arcsin((1 - x)/2) = \pi$$

...
But how can I find all such (m, n) pairs? Since m and n are up to 100, perhaps we can write a program to check for each possible pair (m, n) whether there exists an x in $(0, 1)$ such that the equation is satisfied. But how to implement this check in Python? The problem is that the equation is transcendental and might not have an analytical solution, so numerical methods would be needed. For each (m, n), we can define a function $f(x) = m \cdot \arcsin(x/2) + n \cdot \arcsin((1 - x)/2) - \pi$, and check if there's a root of $f(x) = 0$ in $(0, 1)$, excluding $x = 0.5$. But how to check for roots? For example, we can use the Intermediate Value Theorem. If $f(x)$ changes sign over an interval, there's a root in that interval.
...
Now, let's proceed to write the Python code.
<code>

```python
import math

def has_solution(m, n):
  def f(x):
    return m * math.asin(x/2) + n * math.asin((1 - x)/2) - math.pi

  # Check interval (0, 0.5)
  left = 0.0001
  right = 0.4999
  if f(left) * f(right) < 0:
    return True

  # Check interval (0.5, 1)
  left = 0.5001
  right = 0.9999
  if f(left) * f(right) < 0:
    return True

  return False

count = 0
for m in range(1, 101):
  for n in range(1, 101):
    if has_solution(m, n):
      count += 1

print(count)

```

</code>

<interpreter> 940 </interpreter>

...

Wait, but the code is written in Python, and the interpreter output is 940. So the final answer is $\boxed{940}$.

Pattern 2: Exploration and verification via code. For problems where the solution path is not immediately obvious, the model utilizes the code interpreter as an interactive sandbox for exploration and hypothesis testing. Instead of committing to a *single line* of reasoning, it formulates conjectures and writes short code snippets to test them, observe their outcomes, and iteratively refine its strategy based on the feedback. This pattern is particularly prevalent in problems with low algorithmic amenability, where it allows the model to build confidence and discover insights through empirical experimentation.

Table 8 provides a clear instance of pattern 2: exploration and verification via code. The model first derives a candidate value of $\lambda = \sqrt{3}$ from a simple case, then uses the code interpreter to numerically explore more different scenarios. The feedbacks validate its initial hypothesis and pivot its strategy from further exploration toward constructing a rigorous algebraic proof.

These first two patterns represent a fundamental departure from pure-text reasoning. As we established in Section 2, they constitute entirely new *Computational Equivalence Classes*, new strategies for solving problems. While a pure-text model might theoretically be able to simulate these processes, the token cost of doing so would be astronomical. The step-by-step, trial-and-error nature of the exploratory pattern, in particular, would lead to a blow-up in token length. Therefore, these strategies lie far outside the *Feasible Support under Budget* B for any practical context window, making them uniquely accessible to the TIR paradigm.

Table 8: A case of exploration and verification via code.

[Question]

Find the largest real number λ with the following property: for any positive real numbers p, q, r, s there exists a complex number $z = a + bi(a, b \in \mathbb{R})$ such that

$$|b| \geq \lambda|a| \quad \text{and} \quad (pz^3 + 2qz^2 + 2rz + s) \cdot (qz^3 + 2pz^2 + 2sz + r) = 0.$$

[Response]

Okay, let's try to tackle this problem step by step......

But how do we approach this? Maybe we can think about the problem by considering specific cases ... Perhaps the worst case occurs when $p = q$ and $r = s$? ... Let's try to consider a symmetric case. ... Maybe choosing $p = q = 1$ and $r = s = 1$. Let's try that. ...

The roots are $z = -1$ and roots of $z^2 + z + 1 = 0$, which are $\frac{-1 \pm i\sqrt{3}}{2}$. Let's compute their $|b|/|a|$:

$$z = \frac{-1 \pm i\sqrt{3}}{2} \rightarrow a = -1/2, b = \pm\sqrt{3}/2.$$

So $|b| = \sqrt{3}/2$, $|a| = 1/2$. So $|b|/|a|$ is $\sqrt{3}$.
So for this case, λ can be as large as $\sqrt{3}$. But this is just for this specific case. . . .
This is getting a bit abstract. Maybe we can consider a case where both $P(z)$ and $Q(z)$ have roots with minimal $|b|/|a|$, and find what that value is. . . . when t is very small. Let me try to find its roots numerically for a specific small t. Let me write a Python code snippet to find the roots of $P(z) = z^3 + 2z^2 + 0.2z + 0.1$.

<code>

```python
import numpy as np

# Coefficients for P(z) = z^3 + 2z^2 + 0.2z + 0.1
coeffs = [1, 2, 0.2, 0.1]
roots = np.roots(coeffs)
print(roots)

```

</code>

<interpreter>

```
[-1.92303902+0.j,      -0.03848049+0.22476716j,      -0.03848049-0.22476716j]
```

</interpreter>

For these complex roots, $|b|/|a|$ is $0.2248/0.0385 \approx 5.83$. . . . Wait, but this case gives a high value of $|b|/|a|$. So maybe the worst case is when the minimal such value is low. . . .
But how to find that? It seems challenging without more insight. . . . Alternatively, consider using symmetry. . . . But if we set $s = 4qr/p$, for example, let's set $p = 1, q = 1, r = 1$, then $s = 4 * 1 * 1/1 = 4$. So $P(z) = z^3 + 2z^2 + 2z + 4$. Let's find its roots. Using numpy again:

<code>

```python
import numpy as np

coeffs = [1, 2, 2, 4]
roots = np.roots(coeffs)
print(roots)

```

</code>

<interpreter>

```
[-2.00000000e+00+0.j,    1.66533454e-16+1.41421356j,    1.66533454e-16-1.41421356j]
```

</interpreter>

So $|b|/|a|$ is infinite (since a is nearly zero), but for practical purposes, this suggests that for this case, there are roots with very large $|b|/|a|$. So this case is not helpful. . . . This suggests that for this case, we can achieve $|b|/|a|$ up to $\sqrt{3}$.
. . .
Let's think of a case where the ratio is lower than $\sqrt{3}$. . . . Let me consider the case where $p = q = 1, r = 1, s = 2$. Then $P(z) = z^3 + 2z^2 + 2z + 2$. Let's find its roots using numpy.

<code>

```python
import numpy as np

coeffs = [1, 2, 2, 2]
roots = np.roots(coeffs)
print(roots)

```

</code>

<interpreter>

```
[-1.54368901+0.j,      -0.22815549+1.11514251j,      -0.22815549-1.11514251j]
```

</interpreter>

So $|b|/|a| = 1.115/0.228 \approx 4.89$. The real root has $|b|/|a| = 0$. So again, the maximum is 4.89.
This doesn't seem to help. . . . Let me think of a different approach. . . .
Let me assume that there is a root $z = a + bi$ with $|b| = \lambda|a|$. Let's assume $a \neq 0$. Then, $b = \lambda a$ or $b = -\lambda a$. Let's assume $b = \lambda a$. . . . Substitute $z = a + \lambda ai = a(1 + \lambda i)$ into $P(z) = pz^3 + 2qz^2 + 2rz + s = 0$.
. . .
Since $a \neq 0$ and λ is non-zero (otherwise, we have real roots), we can divide both sides by λa:

$$pa^2(3 - \lambda^2) + 4qa + 2r = 0$$

This is one equation.
From the equation, since all terms except the first are positive, we must have $(3 - \lambda^2) < 0$, which implies $\lambda^2 > 3$, or $\lambda > \sqrt{3}$. This suggests that for *any* non-real root, the ratio $|b|/|a|$ must be greater than $\sqrt{3}$.
...
So, we have two key pieces of information:

1. We found a specific case ($p = q = r = s = 1$) where the largest available ratio is exactly $\sqrt{3}$. This means λ cannot be greater than $\sqrt{3}$.
2. Our general proof shows that for any case with a non-real root, the ratio must be greater than or equal to $\sqrt{3}$. This means λ must be at least $\sqrt{3}$.

Combining these, the largest possible value for λ is exactly $\boxed{\sqrt{3}}$.

Pattern 3: Offloading complex calculation. This is the most direct pattern of tool use, where the model has a clear, linear path to the solution but delegates complex or tedious calculations to the interpreter. This usage aligns with the naive view of TIR as a "calculator", but its importance should not be understated. By offloading these steps, the model minimizes the *risk* of unforced computational errors that frequently derail long chains of pure-text thought, thereby preserving the integrity of the overall reasoning process.

A representative example is shown in Table 9. Here, the model first performs the text-based reasoning to establish a solution path, then uses the interpreter as a precision tool to execute the series of intricate vector and algebraic computations that would be highly prone to manual error.

In conclusion, these emergent patterns reveal a sophisticated interplay between the LLM's reasoning capabilities and the code interpreter's computational power. The model is not merely using a tool; it is *thinking with* tools. This signifies a fundamental shift in strategy: rather than simply delegating calculations from an otherwise unchanged, text-based line of thought, the model learns to generate novel problem-solving approaches that are intrinsically synergistic with the interpreter. It formulates plans that leverage programmatic strengths like iteration and DP from the outset, developing new "Computational Equivalence Classes" that were previously inaccessible. Such dynamic and flexible code invocation enables the TIR model to break the capability ceiling of its pure-text counterpart.

Table 9: A case of offloading complex calculation.

[Question]
Let acute triangle ABC have circumcenter O, and let M be the midpoint of BC. Let P be the unique point such that $\angle BAP = \angle CAM, \angle CAP = \angle BAM$, and $\angle APO = 90°$. If $AO = 53, OM = 28$, and $AM = 75$, compute the perimeter of $\triangle BPC$.

[Response]
Okay, let's try to solve this problem step by step.
...
Alternatively, maybe using coordinate geometry. Let me think about setting up coordinates.
...
First, compute A_x, A_y numerically.

$$A_x = (75\sqrt{13})/7, \qquad A_y = 450/7$$

Then compute vectors AB and AC, their lengths, unit vectors, angle bisector direction, then reflect AM over this direction to get the direction of AP. Then find the intersection of this line with the circle.
But implementing all this in code is time-consuming, but let's try. Let me write the code.
<code>

```python
import math

# Given values
AO = 53
OM = 28
```

```python
AM = 75

# Compute Ay
Ay = 450 / 7
# Compute Ax
sqrt13 = math.sqrt(13)
Ax = (75 * sqrt13) / 7

# Coordinates of A
A = (Ax, Ay)
# Coordinates of O
O = (0, 28)
# Coordinates of M
M = (0, 0)
# Coordinates of B and C
B = (-45, 0)
C = (45, 0)

# Function to compute vector from point1 to point2
def vector(p1, p2):
  return (p2[0] - p1[0], p2[1] - p1[1])

# Compute vectors AB and AC
AB = vector(A, B)
AC = vector(A, C)

# Compute lengths of AB and AC
len_AB = math.hypot(AB[0], AB[1])
len_AC = math.hypot(AC[0], AC[1])

# Compute unit vectors
unit_AB = (AB[0]/len_AB, AB[1]/len_AB)
unit_AC = (AC[0]/len_AC, AC[1]/len_AC)

# Angle bisector direction
angle_bisector_dir = (unit_AB[0] + unit_AC[0], unit_AB[1] + unit_AC[1])

# Vector AM
AM_vec = vector(A, M)

# Function to reflect a vector over another vector
def reflect_vector(v, axis):
  # Normalize the axis vector
  axis_len = math.hypot(axis[0], axis[1])
  if axis_len == 0:
    return v
  axis_unit = (axis[0]/axis_len, axis[1]/axis_len)

  # Dot product
  dot = v[0] * axis_unit[0] + v[1] * axis_unit[1]

  # Projection of v onto axis
  proj = (dot * axis_unit[0], dot * axis_unit[1])

  # Reflection
  reflected = (2*dot*axis_unit[0] - v[0], 2*dot*axis_unit[1] - v[1])
  return reflected

# Reflect AM_vec over angle bisector direction
AP_dir = reflect_vector(AM_vec, angle_bisector_dir)

```

```
# Normalize AP_dir to get direction
AP_dir_len = math.hypot(AP_dir[0], AP_dir[1])
AP_dir_unit = (AP_dir[0]/AP_dir_len, AP_dir[1]/AP_dir_len)

# Parametric line for AP: A + t * AP_dir_unit
# Find intersection with circle with diameter AO

# Equation of circle with diameter AO
# (x - mid_x)^2 + (y - mid_y)^2 = r^2
mid_x = (A[0] + O[0])/2
mid_y = (A[1] + O[1])/2
r = AO / 2

# Parametric equations for line AP
# x = A[0] + t * AP_dir_unit[0]
# y = A[1] + t * AP_dir_unit[1]

# Substitute into circle equation
# (x - mid_x)^2 + (y - mid_y)^2 = r^2
# Expand and solve for t

def solve_for_t():
  # x = A[0] + t * dx
  # y = A[1] + t * dy
  dx = AP_dir_unit[0]
  dy = AP_dir_unit[1]

  # Expand (x - mid_x)^2 + (y - mid_y)^2 = r^2
  # (A[0] + t*dx - mid_x)^2 + (A[1] + t*dy - mid_y)^2 = r^2
  # Expand terms
  a = dx**2 + dy**2
  b = 2 * (dx*(A[0] - mid_x) + dy*(A[1] - mid_y))
  c = (A[0] - mid_x)**2 + (A[1] - mid_y)**2 - r**2

  # Solve a*t^2 + b*t + c = 0
  discriminant = b**2 - 4*a*c
  if discriminant < 0:
    return None
  t1 = (-b + math.sqrt(discriminant)) / (2*a)
  t2 = (-b - math.sqrt(discriminant)) / (2*a)
  return t1, t2

t_values = solve_for_t()
if t_values:
  t = t_values[0]  # Choose one solution, the other might be A itself
  px = A[0] + t * AP_dir_unit[0]
  py = A[1] + t * AP_dir_unit[1]
  P = (px, py)
else:
  P = None

# Now compute perimeter of triangle BPC
if P:
  def distance(p1, p2):
    return math.hypot(p1[0]-p2[0], p1[1]-p2[1])

  BP = distance(B, P)
  CP = distance(C, P)
  BC = distance(B, C)
  perimeter = BP + CP + BC
  print(perimeter)
else:
    print("No intersection found")

```

</code>

<interpreter> 192.0 </interpreter>

The code computes the perimeter of triangle BPC and returns 192.0. Therefore, the answer is $\boxed{192}$.