



SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

**Tool-Integrated Mathematical Language
Model Reasoning with Implicit Process
Reward Models**

Christo Wilken





SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

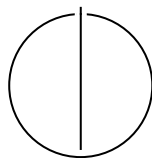
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

**Tool-Integrated Mathematical Language
Model Reasoning with Implicit Process
Reward Models**

**Tool-integrierte mathematische
Sprachmodell-Inferenz mit impliziten
Prozessbelohnungsmodellen**

Author: Christo Wilken
Examiner: Prof. Dr. Stefan Bauer
Supervisor: Prof. Dr. Stefan Bauer
Submission Date: August 08, 2025



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, August 08, 2025

Christo Wilken

Acknowledgments

I would like to thank Prof. Dr. Stefan Bauer for supervising this thesis, for securing the computational resources necessary for this research, and for the trust and autonomy he granted me throughout this project.

Abstract

Large language models struggle with mathematical reasoning tasks that require precise numerical calculation or exhaustive algorithmic computation, often generating thousands of tokens of fruitless reasoning where a few lines of code would provide direct solutions. While tool-integrated reasoning (TIR) addresses these limitations by enabling models to interleave natural language reasoning with code execution, creating high-quality training data for TIR remains a complex and resource-intensive bottleneck.

This thesis presents an efficient approach to enabling tool-integrated mathematical reasoning through systematic data generation, quality filtering, and parameter-efficient fine-tuning. We develop a novel multi-stage pipeline that generates 40,000 initial TIR traces using Gemini 2.0 Flash and aggressively filters them to just 3,410 high-quality examples based on structural heuristics and desirable cognitive behaviors such as verification, subgoal setting, and code-first reasoning. Despite its small size, fine-tuning Qwen3-8B with LoRA on this dataset transforms the model from not exhibiting tool-integrated reasoning behavior to achieving 28.33% accuracy on AIME 2024.

To further improve performance via test-time search, we adapt implicit process reward models to TIR, thereby learning to provide step-level rewards by training only on outcome labeled data. This avoids the prohibitive computational cost incurred by traditional PRM training data generation. Our evaluation on AIME 2025 demonstrates that PRM-guided greedy best-first search at branch factor 8 (28.54%) substantially outperforms unguided solving (21.56%).

Our work validates that high-quality tool-integrated reasoning can be achieved with modest resources through extensive quality filtering, and that implicit PRMs provide effective process supervision for guiding mathematical tool-integrated reasoning at test-time. These contributions demonstrate a path toward efficient and effective tool-integrated reasoning systems.

Contents

Acknowledgments	iv
Abstract	v
1. Introduction	1
1.1. The Challenge to Mathematical Reasoning in LLMs	1
1.2. The Vision of Tool-Integrated Reasoning	2
1.3. Research Questions	2
1.4. Contributions	3
1.5. Thesis Overview	4
2. Background and Related Work	5
2.1. Mathematical Reasoning in LLMs	5
2.1.1. The Journey of Pure Reasoning	5
2.1.2. The Nature of Mathematical Problem Solving	6
2.1.3. Current State and Emerging Directions	7
2.2. Tool-Integrated Reasoning	8
2.2.1. ToRA: Establishing Tool-Integrated Reasoning	8
2.2.2. The Training Data Bottleneck	9
2.3. Process Supervision for Reasoning	10
2.3.1. Outcome versus Process Supervision	11
2.3.2. The Empirical Case for Process Supervision	11
2.3.3. The Annotation Bottleneck	12
2.3.4. Automated Process Supervision: Promise and Limitations	12
2.3.5. The Implicit Revolution	13
2.4. Summary	14
3. Generating High-Quality TIR Training Data	15
3.1. The Data Challenge	15
3.2. Evolution of Our Approach	15
3.2.1. Failed Attempt: Transformation	15
3.2.2. Success: Bottom-up Generation	17

3.3. Dataset Selection	19
3.3.1. DeepScaler: Comprehensive Coverage	19
3.3.2. LIMO: Extreme Challenge as Quality Signal	20
3.4. Prompt Engineering for TIR	20
3.4.1. The Prompt Development Process	21
3.4.2. The Code-First Approach	21
3.5. Multi-Stage Quality Control	24
3.5.1. Stage 1: Heuristic Filtering	25
3.5.2. Stage 2: Behavior-Based Filtering	28
3.6. Data Generation Results	33
3.6.1. TIR Example: Music Program Arrangement with Error Recovery	34
3.7. Summary	40
4. Building the TIR System	41
4.1. Model Selection and Adaptation	41
4.1.1. Base Model Selection	41
4.1.2. LoRA Configuration	42
4.2. System Architecture and TIR Engine	42
4.2.1. Multi-Turn TIR Orchestration	43
4.2.2. Code Execution Framework	43
4.2.3. Answer Verification System	45
4.3. Fine-Tuning Experiments	46
4.3.1. Hyperparameter Experimentation	46
4.3.2. Final Configuration Selection	48
4.4. Deployment Optimization	50
4.4.1. AWQ Quantization Strategy	50
4.4.2. Performance Trade-off Analysis	50
4.5. Summary	51
5. Process Reward Models for TIR	52
5.1. The Test-Time Improvement Challenge	52
5.2. Adapting PRMs to TIR	53
5.3. Generating Training Data	54
5.4. Training the PRM	55
5.4.1. Training Configuration	55
5.4.2. Training Dynamics	55
5.5. Summary	57

6. Evaluation and Results	58
6.1. Evaluation Methodology	58
6.1.1. Benchmark Selection	58
6.1.2. Evaluation Protocol	58
6.2. Evaluation Modes	59
6.2.1. Direct Solving (Baseline)	59
6.2.2. Majority Voting	59
6.2.3. Greedy Best-First Search with PRM Guidance	60
6.3. Results	61
6.3.1. AIME 2025 Performance	61
6.3.2. Scaling Analysis	62
6.3.3. Computational Efficiency	63
6.4. Discussion	63
6.5. Summary	64
7. Discussion and Insights	65
7.1. Key Findings	65
7.1.1. Efficient TIR Training is Achievable	65
7.1.2. PRM Warmup Improves Discrimination	65
7.1.3. Test-Time Scaling Behavior	66
7.2. Implementation Guidance	66
8. Conclusion and Future Work	68
8.1. Summary of Contributions	68
8.2. Limitations	69
8.2.1. No Systematic Ablations	69
8.2.2. Scale Constraints	69
8.2.3. Computational Analysis	70
8.2.4. Limited Contemporary Benchmarking	70
8.3. Future Work	71
8.3.1. Immediate Extensions	71
8.3.2. Evaluation Improvements	71
8.3.3. Longer-term Directions	72
8.4. Final Thoughts	72
A. Prompt Templates for Data Transformation	73
A.1. Code-Based Approach Creation	73
A.2. Code Checkpoint Creation	77
A.3. Combining Pure Math with Code	81

B. Prompt Template for TIR Generation	90
B.1. Code First Math Solver	90
Abbreviations	98
List of Figures	99
List of Tables	101
Bibliography	102

1. Introduction

1.1. The Challenge to Mathematical Reasoning in LLMs

Large language models (LLMs) have transformed numerous domains of artificial intelligence, achieving strong performance on tasks ranging from natural language understanding to code generation. Yet when confronted with mathematical reasoning, even advanced models exhibit limitations. This discrepancy becomes particularly apparent when we examine how these models handle problems that require precise numerical calculation or exhaustive algorithmic computation.

Consider a seemingly simple task: finding the 10th palindromic prime number between 10,000 and 99,999. State-of-the-art reasoning models like DeepSeek-R1 [Dee25], despite their advanced chain-of-thought capabilities and extended reasoning processes, struggle with this problem. Such models can generate thousands of tokens of reasoning yet fail to arrive at the correct answer. This failure is particularly instructive because the problem admits a straightforward computational solution: a simple Python script of fewer than 20 lines can solve it in milliseconds.

This example illustrates two key challenges to mathematical reasoning in LLMs. First, there is the *calculation accuracy problem*: models trained on natural language struggle with precise numeric operations and algorithmic computation. While they may understand the conceptual approach to a problem, executing the necessary calculations reliably remains difficult. Second, there is the *long reasoning chains issue*: as models attempt to work through complex problems using pure natural language reasoning, they often become trapped in circular logic, lose track of intermediate results, or accumulate errors that compound throughout the reasoning process.

The limitations extend beyond simple computational tasks. Until very recently, even the most advanced models struggled on challenging mathematical benchmarks like the AIME (American Invitational Mathematics Examination)¹. While the latest frontier models have made impressive strides, achieving human-competitive performance has required either massive scale, or, as we explore in this thesis, effective tool integration.

¹The AIME is a selective 15-question, 3-hour examination for high school students who rank in the top 5% on the American Mathematics Competitions.

1.2. The Vision of Tool-Integrated Reasoning

In response to these challenges, a new paradigm has emerged: Tool-Integrated Reasoning (TIR). Rather than forcing language models to perform all computation through natural language, TIR enables models to interleave reasoning with external tool use, particularly code execution. This approach mirrors how many mathematicians and scientists work in practice, alternating between high-level reasoning and computational verification, using calculators or computer algebra systems to handle routine calculations while focusing their attention on problem structure and strategy.

Microsoft’s ToRA (Tool-integrated Reasoning Agent) established a foundation for tool-integrated mathematical reasoning by demonstrating significant improvements across mathematical benchmarks by teaching models when and how to invoke computational tools [Gou+24]. This approach has since become state-of-the-art, as the winning solution of the AI Mathematical Olympiad Progress Prize 2 (AIMO-2) competition² achieved its strong performance through effective tool integration [Mos+25]. By combining natural language reasoning with systematic code execution, the winning team solved 34 out of 50 highly challenging problems, a feat that could not be matched by models relying purely on natural language reasoning.

With tool-integrated reasoning established, we are now faced with new opportunities and challenges. Tool integration expands the solution search space to include both reasoning steps and computational actions, calling for methods for intelligent exploration. Process Reward Models (PRMs) offer a principled approach to navigate this expanded solution space by evaluating the promise of solutions at each step, unlike Outcome Reward Models (ORMs) that can only evaluate final answers. Recent work has shown that process supervision significantly outperforms outcome supervision for mathematical reasoning [Lig+23], but traditional PRM training methods like Math-Shepherd require computationally expensive Monte Carlo rollouts to generate training data [Wan+24].

1.3. Research Questions

This thesis addresses two primary research questions at the intersection of tool-integrated reasoning and process reward models:

RQ1: How can we generate high-quality training data for tool-integrated mathematical reasoning efficiently? Creating effective TIR training data presents unique challenges beyond traditional mathematical datasets. The data should demonstrate

²The AIMO Prize is a \$10 million challenge funded by XTX Markets to develop AI capable of winning a gold medal at the International Mathematical Olympiad. Progress prizes reward intermediate breakthroughs, with competitions hosted on Kaggle. See <https://aimoprize.com>.

proper tool usage patterns, show meaningful interleaving of reasoning and computation, and maintain correctness of both natural language explanations and code outputs. We investigate multi-stage generation and filtering approaches to produce training data that enables effective learning of tool-integrated reasoning capabilities.

RQ2: Can process reward models improve tool-integrated reasoning performance at test time? While PRMs have shown promise for pure reasoning tasks, their application to tool-integrated reasoning introduces new complexities. We explore whether PRMs can effectively guide search through the expanded action space that includes both reasoning steps and tool invocations, and how PRM-guided search compares to simpler test-time strategies such as majority voting for tool-integrated mathematical problem solving.

1.4. Contributions

This thesis makes three primary contributions to the field of mathematical reasoning in language models:

1. A novel multi-stage data generation pipeline for high-quality TIR training data. We develop a systematic approach that generates tool-integrated reasoning traces and applies progressive filtering through multiple stages. Our pipeline employs both heuristic filtering and behavior-based filtering inspired by cognitive science research, ensuring that retained examples demonstrate key reasoning behaviors including verification, subgoal setting, and effective tool usage. This quality-over-quantity approach produces training data that enables strong tool-integrated reasoning capabilities despite its modest size.

2. Empirical validation of our data generation approach through transforming a base model with no TIR capability to achieve 28.33% accuracy on AIME 2024. The base Qwen3-8B model does not exhibit tool-integrated reasoning behavior even when explicitly prompted, defaulting instead to pure mathematical reasoning without code generation. Through fine-tuning on our carefully curated dataset, we enable the model to successfully interleave natural language reasoning with code execution, effectively solving competition-level mathematics problems. This transformation demonstrates that high-quality training data can impart complex capabilities even with limited examples.

3. The first application of implicit process reward models to tool-integrated reasoning, with evaluation on AIME 2025. We adapt the implicit PRM framework [Yua+24] to the unique challenges of TIR, avoiding the substantial computational overhead that traditional Monte Carlo-based approaches require for generating PRM training data. Our adaptation involved defining TIR steps as complete tool-use cycles

(reasoning, code, output) rather than individual sentences, enabling the PRM to evaluate meaningful units of progress. Through comparative evaluation on AIME 2025, we demonstrate that test-time PRM-guided greedy best-first search provides performance competitive with majority voting, validating the applicability of implicit PRMs to tool-integrated reasoning tasks.

1.5. Thesis Overview

The remainder of this thesis is organized as follows:

Chapter 2 provides background on mathematical reasoning in LLMs, tool-integrated reasoning approaches, and process supervision methods.

Chapter 3 presents our data generation pipeline for creating high-quality tool-integrated reasoning training data.

Chapter 4 describes the construction of our TIR system, including model selection, fine-tuning experiments, and deployment optimization.

Chapter 5 details the development of process reward models adapted for tool-integrated reasoning.

Chapter 6 evaluates our approach on AIME 2025 and compares different test-time inference strategies.

Chapter 7 discusses key findings and their implications for research and practice.

Chapter 8 concludes with a summary of contributions, limitations, and future work.

2. Background and Related Work

The challenges identified in Chapter 1 emerge from a rich history of research in mathematical reasoning and tool use in language models. This chapter provides comprehensive background on three interconnected research threads that inform our approach: the evolution of mathematical reasoning capabilities in LLMs, the emergence of tool-integrated reasoning, and the development of process supervision methods that enable more reliable multi-step reasoning. By examining prior work in these areas, we establish both the theoretical foundations and practical constraints that shape our contributions in subsequent chapters.

2.1. Mathematical Reasoning in LLMs

Mathematical reasoning represents one of the most intriguing challenges in artificial intelligence, requiring the integration of abstract thinking, logical deduction, and precise computation. For large language models, this domain has served as both a guiding benchmark and a driving force for innovation. The journey from early struggles with elementary arithmetic to achieving human-competitive performance on competition mathematics reveals not just the significant progress of these models, but also key questions about how artificial systems should approach mathematical problem-solving in the future.

2.1.1. The Journey of Pure Reasoning

Early work on mathematical reasoning in language models revealed that even grade school mathematics posed significant challenges. Cobbe et al. [Cob+21] introduced GSM8K, a dataset of 8,500 linguistically diverse grade school math word problems, and found that contemporary large language models struggled to achieve high accuracy despite the conceptual simplicity of the problems. This work established an important baseline, demonstrating that mathematical reasoning required more than simply scaling up model size or training data.

A significant milestone came with Minerva [Lew+22], which represented an early serious attempt to tackle advanced mathematical reasoning through specialized pre-training. By training on a large corpus of mathematical content including papers and

web pages with mathematical notation, Minerva achieved then state-of-the-art performance on technical benchmarks without external tools. On the undergraduate-level MATH dataset [Hen+21], Minerva reached 33.6% accuracy, while on GSM8K it achieved 58.8%. These results demonstrated that targeted training could significantly improve mathematical capabilities, yet the model still struggled with problems requiring lengthy reasoning chains or precise computation.

The evolution continued through various iterations of frontier models, each pushing the boundaries further. A particularly significant improvement came with the recent development of extended reasoning models. DeepSeek-R1 [Dee25], through its reinforcement learning approach that incentivizes step-by-step reasoning, achieved 79.8% accuracy on AIME 2024. This performance approaches that of the strongest high school mathematics competitors.

2.1.2. The Nature of Mathematical Problem Solving

Despite these impressive achievements, a closer examination reveals core characteristics about how language models approach mathematics that illuminate both their strengths and limitations. Mathematical problem-solving inherently involves two distinct types of cognitive work: high-level strategic reasoning and mechanical computation. The former encompasses understanding problem structure, identifying relevant theorems or techniques, and planning solution approaches. The latter involves executing calculations, manipulating algebraic expressions, and verifying numerical results.

Language models, by their nature, excel at the strategic aspects. They can recognize problem patterns, suggest appropriate solution methods, and articulate reasoning steps with excellent clarity. However, when it comes to mechanical computation, they face an inherent mismatch. Consider the example from our introduction: finding the 10th palindromic prime number between 10,000 and 99,999. A human programmer would immediately recognize this as a straightforward algorithmic task, solvable with a simple loop that checks palindromes for primality. The resulting code might be 15-20 lines and execute in milliseconds.

Yet when DeepSeek-R1 or similar extended reasoning models approach this problem, they often generate thousands or even tens of thousands of tokens of reasoning. They attempt to "simulate" the computation through natural language, checking individual numbers, verifying palindrome properties, testing primality through trial division, all expressed verbosely in text. This is not a failure of intelligence but rather an inherent inefficiency: these models are using a linguistic medium to perform what is essentially algorithmic computation.

This phenomenon becomes especially pronounced with problems involving complex calculations, symbolic manipulation, or iterative procedures. The models must main-

tain precision across lengthy reasoning chains, track intermediate results without the benefit of variable storage, and perform arithmetic through pattern matching rather than direct computation. While recent models have become quite effective at this simulation, generating correct results through sheer persistence and careful reasoning, the computational cost is enormous compared to direct code execution.

2.1.3. Current State and Emerging Directions

The current landscape of mathematical reasoning in LLMs thus presents a paradox. On the one hand, pure reasoning approaches have achieved results that would have seemed impossible just a few years ago. DeepSeek-R1’s performance on AIME demonstrates that language models can, through extended reasoning alone, solve problems requiring deep mathematical insight. The quality of mathematical explanations these models can produce can rival those of human experts, with clear step-by-step derivations and insightful problem analysis.

On the other hand, the computational inefficiency of pure reasoning approaches raises important questions about the optimal architecture for mathematical problem-solving systems. The thousands of tokens required to solve computationally straightforward problems represent not just an inefficiency but a fundamental mismatch between the tool and the task. This observation has led researchers to explore alternative paradigms that better align with the dual nature of mathematical work.

Notably, recent competitions have begun to validate these alternative approaches. The AI Mathematical Olympiad Progress Prize 2 (AIMO-2), which specifically targets competition-level mathematics, saw its winning solution by Moshkov et al. [Mos+25] employ a markedly different strategy from pure reasoning. Rather than relying solely on extended chains of natural language reasoning, the winning approach integrated external computational tools, achieving superior results through a more natural separation of strategic reasoning and mechanical computation.

This emergence of tool-integrated approaches as competitive alternatives to pure reasoning sets the stage for a deeper examination of how language models can be augmented with external capabilities. The question is no longer whether LLMs can solve mathematical problems, but rather how they can solve them most effectively and efficiently. This leads us naturally to examine tool-integrated reasoning in detail, exploring how the combination of language models’ strategic reasoning capabilities with precise computational tools might offer a more promising path forward.

2.2. Tool-Integrated Reasoning

2.2.1. ToRA: Establishing Tool-Integrated Reasoning

Given the computational inefficiencies of pure reasoning approaches discussed above, researchers have explored integrating external computational tools directly into the reasoning process.

Gou et al. [Gou+24] introduced ToRA (Tool-integrated Reasoning Agent), a groundbreaking approach that seamlessly integrates natural language reasoning with external computational tools. Rather than treating language models and computational tools as separate entities, ToRA enabled models to interleave natural language thought processes with code generation and execution, creating a unified reasoning framework.

The key innovation of ToRA lies in its treatment of code as an extension of the reasoning process rather than a separate modality. When solving a mathematical problem, ToRA follows a structured trajectory format: the model generates natural language rationales (r_i) to analyze the problem structure and identify solution strategies, then produces program actions (a_i) at points where computation is beneficial, executes the code to obtain outputs (o_i), and incorporates these results back into subsequent reasoning. This creates trajectories of the form $\tau = r_1 a_1 o_1 \dots r_{n-1} a_{n-1} o_{n-1} r_n$, where the model alternates between generating reasoning/code and receiving execution outputs, proceeding autonomously through the problem without external intervention. The seamless interleaving of natural language and code execution (typically using Python with libraries like SymPy for symbolic mathematics) ideally creates a synergistic loop where linguistic analysis guides computational exploration, and computational results inform strategic decisions.

The impact of this approach is significant. On the MATH dataset [Hen+21], ToRA-7B achieved 44.6% accuracy, surpassing the much larger WizardMath-70B model by approximately 22 percentage points. ToRA-CODE-34B became the first open-weight model to exceed 50% on MATH [Hen+21] at the time, achieving 50.8% and outperforming GPT-4’s Chain-of-Thought (CoT) result of 42.5%, nearly matching GPT-4 with code at 51.8%. Across ten mathematical reasoning benchmarks, ToRA models showed notable and consistent improvements over pure reasoning approaches at every model scale from 7B to 70B. These results demonstrated that tool integration could compensate for model size limitations, enabling smaller models to outperform much larger ones that rely solely on internalized knowledge.

The ToRA training methodology involves two key phases. First, imitation learning on 16,000 high-quality tool-integrated trajectories (ToRA-Corpus) collected from GPT-4 annotations on GSM8K and MATH [Hen+21] datasets. Second, and critically, output space shaping addresses the limitation that curated data cannot exhaust all valid

solution trajectories. This technique trains models on both self-sampled valid trajectories and invalid ones that have been corrected by a teacher model, significantly improving reasoning diversity and enabling models to explore plausible trajectories beyond those seen in training. This two-phase approach proved essential for achieving competitive performance.

The ToRA architecture consists of three key components: (1) a language model fine-tuned using the above methodology, (2) a code execution environment with pre-imported mathematical libraries, and (3) an output integration mechanism that formats execution results for model consumption. During inference, the model generates a reasoning trajectory that may include multiple code blocks, alternating between model generation and code execution with state preserved between executions. This design allows for incremental problem-solving where later computations can build upon earlier results, all within a single autonomous problem-solving session.

While ToRA laid the foundation, the field continues to evolve with advanced training methods and search strategies, though the core principle of interleaving natural language reasoning with code execution remains central.

2.2.2. The Training Data Bottleneck

Despite these architectural advances in tool-integrated reasoning, the field faces a critical bottleneck that constrains further progress: the scarcity of high-quality training data. Unlike pure mathematical reasoning, where models can be trained on human-written solutions or even synthetic data generated by other models, tool-integrated reasoning traces have unique requirements that make data collection particularly challenging.

The core challenge stems from the nature of tool-integrated trajectories. Each training example must contain not only natural language reasoning and code but also the actual outputs from code execution. These outputs cannot be approximated or generated by the model; they must be the result of real code execution to maintain consistency. When a model trained on incorrect outputs encounters real execution results during deployment, the mismatch leads to degraded performance and confusion, a phenomenon we experienced firsthand in our transformation attempts (Section 3.2.1). This "output authenticity" requirement means that every training example must be dynamically generated with actual code execution, significantly increasing the complexity of data creation.

The annotation challenge is compounded by quality requirements. Effective tool-integrated reasoning traces must demonstrate appropriate decisions about when to use tools versus reasoning, how to structure code for clarity and correctness, and how to interpret results meaningfully. Poor quality traces (such as those that use code unnecessarily, write inefficient solutions, or misinterpret outputs) can degrade model

performance. This creates a quality-quantity tradeoff where a smaller set of high-quality examples might outperform a larger set of mediocre ones.

Current approaches to generating TIR training data fall into three categories, each with significant limitations. Manual annotation by human experts produces high-quality traces but is prohibitively expensive and slow, limiting datasets to thousands rather than millions of examples. Automated generation using existing models can produce volume but often results in repetitive patterns and may propagate errors. Conversion of existing pure-math solutions to tool-integrated versions faces the core problem of generating authentic execution outputs without actually running the code.

The scale of data required adds another dimension to the challenge. While Gou et al. [Gou+24] achieved strong results with carefully curated datasets, pushing performance further requires not just more data but more diverse data covering different problem types, difficulty levels, and solution strategies. The AIMO-2 winning solution’s creation of 1.7 million examples [Mos+25] demonstrates the scale needed for state-of-the-art performance, but also highlights the massive computational investment required.

This data bottleneck represents the primary obstacle to democratizing tool-integrated reasoning. While the architectural innovations of ToRA and its successors have shown the path forward, realizing the full potential of TIR requires solving the data generation challenge. This motivates the development of efficient, scalable methods for creating high-quality tool-integrated reasoning traces, a challenge we address in the following chapter through our novel multi-stage data generation pipeline.

2.3. Process Supervision for Reasoning

The practical deployment of tool-integrated reasoning systems hinges on more than just training capable models. During inference, these systems must navigate complex solution spaces where each step combines reasoning with code execution, and early choices can determine success or failure. Without mechanisms to evaluate partial solutions, we must resort to greedy generation or naive sampling, missing opportunities to explore more promising paths. Process supervision addresses this need by providing quality assessments at each reasoning step, but traditional approaches require expensive stepwise annotation of training data. This section examines how implicit process reward models overcome these limitations, providing the benefits of process supervision without requiring prohibitively expensive annotations.

2.3.1. Outcome versus Process Supervision

Traditional approaches to training reasoning models have relied on outcome supervision, where training signal is provided based solely on whether a model reaches the correct final answer. This approach has an appealing simplicity: for mathematical problems with deterministic answers, checking correctness is straightforward and can be fully automated.

However, outcome supervision suffers from inherent limitations that become increasingly problematic as problems grow more complex. When we receive only a binary signal about the incorrectness of a final answer during training, we lack information about where errors occurred in a reasoning chain. This sparse feedback makes exact credit assignment impossible. More troublingly, outcome supervision might reward solutions that reach correct answers through invalid reasoning, a phenomenon particularly common in mathematical domains where multiple errors might coincidentally cancel out.

Process supervision offers a different approach. Rather than evaluating only the final result, a process reward model (PRM) assigns rewards to each intermediate step in the reasoning chain. This dense feedback provides precise error localization, enabling models to learn which types of reasoning steps lead to successful outcomes. The benefits can extend beyond training efficiency: process supervision can promote more interpretable and aligned behavior, as models can be rewarded for following human-endorsed reasoning patterns rather than simply reaching correct answers through any means.

2.3.2. The Empirical Case for Process Supervision

The theoretical advantages of process supervision were empirically validated by Lightman et al. [Lig+23] in their landmark study "Let's Verify Step by Step." Through extensive experimentation on the MATH dataset [Hen+21], they demonstrated that process supervision not only matches but significantly outperforms outcome supervision in training reliable mathematical reasoning models. Their process-supervised model achieved 78.2% accuracy on a representative subset of MATH, compared to 72.4% for the outcome-supervised variant, despite both models using identical architectures and training procedures.

The superiority of process supervision manifested across multiple dimensions. In best-of-N reranking scenarios, where models select the best solution from multiple candidates, PRMs consistently outperformed ORMs at every value of N tested. The gap widened with larger N values, suggesting that PRMs better distinguish between subtly different reasoning paths. More importantly, analysis of model outputs revealed

that process-supervised models exhibited more consistent reasoning patterns, making fewer logical errors even when reaching incorrect final answers.

Lightman et al. also introduced the concept of a "negative alignment tax," arguing that process supervision simultaneously improves both performance and alignment. Unlike many machine learning interventions where safety and capability improvements trade off against each other, process supervision enhances model reliability while making behavior more interpretable and controllable. This dual benefit makes process supervision particularly attractive for deployment in high-stakes applications where understanding model reasoning is essential.

While this work demonstrated the power of process supervision during training, our thesis explores how to obtain similar benefits at test time.

2.3.3. The Annotation Bottleneck

Despite these compelling advantages, process supervision faces a severe practical limitation: the cost of obtaining step-level annotations. To train their PRM, Lightman et al. employed human annotators to label each step in model-generated solutions as correct, incorrect, or neutral. This labeling process required mathematical expertise, as annotators needed to evaluate not just computational correctness but also the reasonableness of each reasoning step. The resulting PRM800K dataset contains 800,000 step-level labels across 75,000 solutions, representing an enormous investment in human annotation effort.

The challenges of manual annotation extend beyond mere cost. Maintaining consistency across annotators requires extensive training and calibration, particularly for nuanced judgments about what constitutes a "reasonable" versus "unreasonable" step. Edge cases abound: steps might be technically correct but misleading, or contain minor errors that don't affect the overall solution trajectory. The need for mathematical expertise further constrains the annotator pool, driving up costs and limiting scalability. For many researchers and organizations, the expense of collecting process supervision data at scale can be prohibitive.

2.3.4. Automated Process Supervision: Promise and Limitations

Recognizing these limitations, Wang et al. [Wan+24] proposed Math-Shepherd, an approach to automatically generate process supervision training data without human annotation. Their key insight was to estimate step quality through Monte Carlo Tree Search (MCTS), defining a step as high-quality if it has high potential to lead to correct final answers. For each intermediate step in a solution, Math-Shepherd uses a "completer" model to generate N continuation trajectories, then labels the step based

on what fraction of these trajectories reach the correct answer.

This automated approach showed promising results. Math-Shepherd achieved substantial improvements over baseline models, with Mistral-7B improving from 77.9% to 84.1% on GSM8K and from 28.6% to 33.0% on MATH [Hen+21] when using step-by-step PPO-based training. The ability to generate process supervision without human intervention seemed to offer a path toward democratizing PRM training.

However, Math-Shepherd’s MCTS approach introduces its own severe limitation: computational cost. To generate labels for a single solution with 10 reasoning steps, using 8 continuation trajectories per step as recommended (the setting used in Math-Shepherd’s implementation), the method requires generating 80 complete solution trajectories. This represents approximately 38.8x the computational cost of standard outcome supervision.¹ The quality of annotations also depends critically on the completer model’s capabilities; weak completers may assign poor labels to actually good steps simply because they cannot successfully continue from them. Furthermore, the method suffers from inherent noise in its estimates, as the limited number of sampled trajectories may not accurately represent the true probability of success from each step.

2.3.5. The Implicit Revolution

The seemingly intractable trade-off between annotation cost and computational overhead was elegantly addressed by Yuan et al. [Yua+24] through a theoretical insight about the nature of reward models. Their key observation was that when outcome rewards are parameterized as log-likelihood ratios between policy and reference models (a common practice in preference learning algorithms like DPO), the model implicitly learns a Q-function that can be used for process supervision.

Specifically, by parameterizing the reward as $r_\theta(\mathbf{y}) = \beta \log \frac{\pi_\theta(\mathbf{y})}{\pi_{\text{ref}}(\mathbf{y})}$, where π_θ is the policy model and π_{ref} is a reference model, the cumulative reward up to step t becomes an exponentially-weighted average of future outcome rewards. This means that training a standard outcome reward model with this parameterization automatically yields a process reward model, with no additional data collection or training required. The process reward for each step can be computed as the difference in cumulative rewards, providing dense feedback throughout the reasoning chain.

The empirical results validate this theoretical insight decisively. Yuan et al.’s implicit PRM outperformed Math-Shepherd while requiring less than 1/38 of the computational cost, achieving superior best-of-N performance across multiple model scales. The approach works with various training objectives beyond DPO, including KTO, NCA,

¹Computed as 80 trajectories \div 2.06 trajectories per problem, where 2.06 is the empirical average reported for CE-based outcome supervision in Yuan et al. [Yua+24].

and even simple cross-entropy loss, making it broadly applicable. Notably, training on additional step-level labels from Math-Shepherd brought no further improvements to the implicit PRM, suggesting that the automatically derived process rewards capture the essential signal without explicit supervision.

The implications for tool-integrated reasoning are particularly significant. As discussed in the previous section, TIR already faces substantial data generation challenges. Adding traditional process supervision would compound these challenges, requiring expensive step-level annotations for traces that include code execution outputs. Implicit PRMs elegantly sidestep this complexity, enabling us to obtain process supervision benefits from standard outcome-labeled rollouts without the high computational overhead of Monte Carlo methods. This makes test-time process supervision practical for TIR systems, as we demonstrate in the following chapters through our implementation of implicit PRMs for tool-integrated mathematical reasoning.

2.4. Summary

This chapter established the theoretical and practical foundations for our work on tool-integrated mathematical reasoning with process reward models. We traced the evolution of mathematical reasoning in LLMs from early struggles with grade-school problems to recent achievements on competition mathematics, highlighting the central tension between impressive capabilities and computational inefficiency. We examined how tool-integrated reasoning emerged as a solution, with ToRA demonstrating the power of interleaving natural language with code execution, while also revealing the critical bottleneck of training data scarcity. Finally, we explored the promise and challenges of process supervision, showing how implicit PRMs offer a path to practical deployment by eliminating the substantial computational overhead of traditional approaches. These three research threads (mathematical reasoning, tool integration, and process supervision) converge in our work to enable scalable, efficient, and effective mathematical problem-solving systems.

3. Generating High-Quality TIR Training Data

3.1. The Data Challenge

The AIMO-2 competition winners [Mos+25] demonstrated the power of scale in tool-integrated reasoning, using 1.7 million tool-integrated reasoning traces to achieve state-of-the-art results. This massive dataset represents the current pinnacle of open-source TIR data generation, combining careful prompt engineering with extensive computational resources to create the OpenMathReasoning dataset.

However, LIMO’s recent success [Ye+25] in training high-performance natural language reasoning models using just 817 carefully curated problems raised an interesting question: could a similar small-scale quality-first approach work for tool-integrated reasoning? While Moshkov et al. pursued scale, we explored the opposite direction: applying aggressive quality filtering to create a compact, behavior-optimized TIR dataset.

This exploratory approach allowed us to investigate whether LIMO’s “less is more” philosophy could translate to the TIR domain, where combining natural language reasoning with code execution creates additional quality assessment challenges. Our work represents a systematic attempt to apply principled quality filtering designed for the unique requirements of tool-integrated mathematical reasoning.

We detail our journey from initial attempts at data generation through the development of a systematic multi-stage filtering pipeline. Starting with approximately 40,000 generated traces, we retained only the 27,826 correct solutions (69.6%), then filtered these down to 3,410 high-quality examples. This 91.5% reduction from the initial pool yielded a dataset that successfully trained TIR models.

3.2. Evolution of Our Approach

3.2.1. Failed Attempt: Transformation

One intuitive approach to generating tool-integrated reasoning training data appeared to be leveraging existing high-quality pure mathematical reasoning datasets. Datasets

like LIMO [Ye+25] and others contain carefully curated mathematical problems paired with detailed step-by-step pure mathematical solutions (natural language reasoning without any code execution), where solutions are systematically selected from mixed human expert and AI-generated sources through expert curation and quality assessment. Rather than discarding this valuable resource, we initially attempted to transform these pure reasoning traces into tool-integrated format through a multi-step pipeline.

Our transformation pipeline employed a two-stage approach designed to analyze these code-free mathematical solutions from multiple perspectives and synthesize them into tool-integrated reasoning traces. In the first stage, we generated two complementary approaches in parallel: a “code-based approach” that restructured the mathematical solution with explicit code verification steps, and a “checkpoint-based approach” that identified key validation points where code could verify intermediate results. These approaches analyzed the pure mathematical solution from different angles, the code-based approach completely restructuring solutions to lead with computational discovery, and the checkpoint-based approach preserving the original mathematical flow while inserting strategic code blocks to confirm intermediate calculations. In the second stage, we combined both generated approaches with the original pure mathematical solution, asking a language model to synthesize all three inputs into a flowing narrative that naturally interleaved reasoning with code execution. The specific prompts used for these transformation approaches are included in Appendix A: the code-based approach (Appendix A.1), the checkpoint-based approach (Appendix A.2), and the synthesis prompt that combines them (Appendix A.3).

The limitations of this approach became clear when we attempted to validate the generated traces. Our pipeline required models to predict what the standard output (stdout) would be for each code block without actually executing the code. While language models have become highly capable at understanding code logic, predicting exact execution outputs remains inherently unreliable. Even advanced models like Claude 3.5 Sonnet, which we employed for the transformation, could correctly understand that a loop would find prime numbers, but could not predict exactly which primes would be found in a specific range, or the precise floating-point result of a complex calculation. When we validated these predictions against actual execution results, we found mismatches throughout the generated dataset. These mismatches were not only cosmetic, but represented critical inconsistencies in the resulting training data.

We found that a model trained on such data would learn flawed patterns: it would ignore computational evidence that contradicts its reasoning or expect outputs that differ from actual execution results. The approach failed because tool-integrated reasoning training data must contain authentic execution outputs, not predictions.

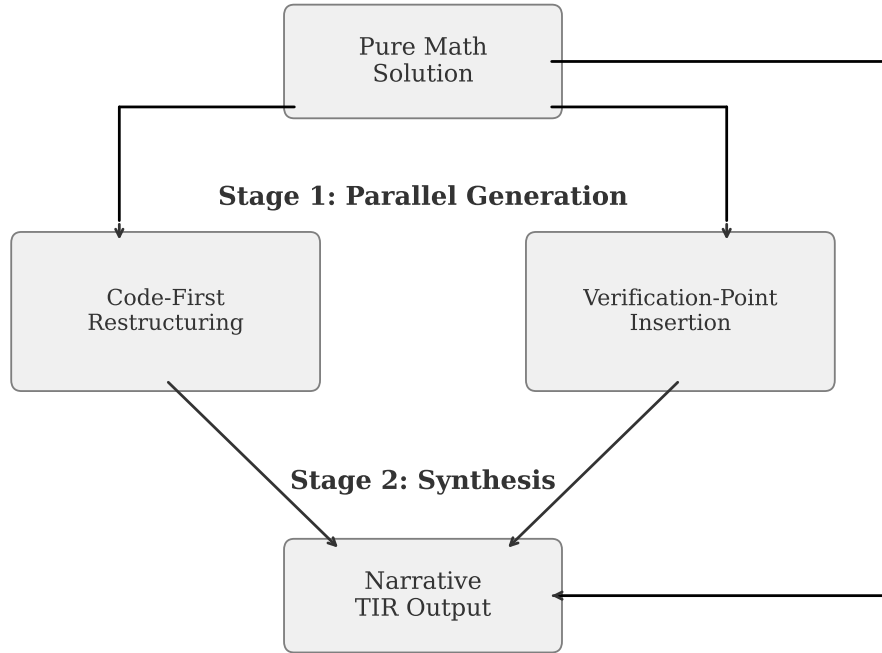


Figure 3.1.: Failed transformation pipeline for converting pure mathematical solutions to tool-integrated reasoning traces. Stage 1 generates approaches with *predicted* code outputs, Stage 2 synthesizes them.

3.2.2. Success: Bottom-up Generation

Recognizing that transformation could not produce authentic tool-integrated reasoning traces, we pivoted to a different approach: generating new traces from scratch with real code execution integrated throughout the process. Rather than attempting to retrofit existing solutions with predicted outputs, we would have models solve problems using tool-integrated reasoning from the beginning, capturing genuine execution results as they were generated.

For this bottom-up generation approach, we selected Gemini 2.0 Flash as our primary model. This choice was driven by Gemini 2.0 Flash’s unique position on the cost-performance frontier for large-scale generation tasks. The model offers highly competitive pricing per token while maintaining strong mathematical reasoning capabilities, making it economically feasible to generate tens of thousands of traces. Additionally, its fast inference times and high rate limits enabled efficient parallel processing, al-

lowing us to attempt many problems simultaneously without encountering throttling issues. Despite being optimized for efficiency, Gemini 2.0 Flash demonstrated sufficient mathematical reasoning capabilities to successfully employ tool-integrated reasoning on competition-level problems.

The technical implementation of our bottom-up approach centers on a multi-turn conversation system that maintains persistent code execution state throughout the problem-solving process. When Gemini 2.0 Flash generates a reasoning step that includes Python code, our system immediately executes that code with NumPy and SymPy pre-imported. The detailed implementation of this TIR orchestration process is discussed in Section 4.2.1. The actual output from this execution, whether stdout or stderr, is captured and appended to the conversation as a user message. This creates a dialogue where the model can see and respond to real execution results, adjusting its reasoning and next steps accordingly.

Through this approach, we conducted a large-scale generation run using problems sourced from the DeepScaler [Luo+25] and LIMO [Ye+25] datasets. We discuss the rationale for these choices in Section 3.3. We chose this scale to provide enough volume for aggressive quality filtering while keeping computational costs manageable. Rather than pursuing maximum scale, we focused on generating enough high-quality raw material to support our multi-stage filtering pipeline, which would ultimately distill these traces down to the highest-quality examples.

The generation process yielded approximately 40,000 initial traces, of which 27,826 (69.6%) successfully solved their respective problems with correct final answers. Only these correct solutions proceeded to our quality filtering pipeline, as training on incorrect traces would teach models flawed reasoning patterns regardless of their other qualities.

While bottom-up generation solved the problem of output authenticity, it introduced a new challenge: both the quality and success rate of generated traces varied significantly depending on how we prompted the model to approach problems in a tool-integrated manner. Without carefully crafted prompts, models would default to pure mathematical reasoning or use code ineffectively, leading to low success rates even on problems well within their capabilities. Our experiments revealed that prompt engineering could improve success rates by over 20 percentage points on the same problem set (from 36.67% with pure mathematical reasoning to 56.67% with effective tool-integrated reasoning on AIME 2024, as we detail in Section 3.4). Maximizing the effectiveness of our bottom-up generation approach would require careful prompt engineering to guide Gemini 2.0 Flash toward both correct solutions and high-quality tool-integrated reasoning patterns.

3.3. Dataset Selection

The choice of source problems substantially affected the quality and diversity of generated traces. We selected two complementary datasets: DeepScaler [Luo+25] for comprehensive coverage and LIMO [Ye+25] for extreme difficulty. This combination provided both the volume needed for aggressive filtering and the problem diversity necessary for robust training.

Table 3.1.: Source datasets for tool-integrated reasoning generation. DeepScaler provides breadth through comprehensive coverage, while LIMO offers depth through extreme-difficulty problems.

Dataset	# Problems	Description
DeepScaler	40,315	Competition problems from AIME (1984-2023), AMC, Omni-MATH, and Still dataset; pre-filtered for quality
LIMO	817	Extreme difficulty problems curated from millions of candidates to challenge state-of-the-art models
Total	41,132	<i>Combined dataset balances broad coverage with targeted difficulty</i>

3.3.1. DeepScaler: Comprehensive Coverage

DeepScaler proved well-suited for large-scale TIR generation, providing 40,315 mathematics-specific questions drawn from prestigious competitions. The dataset aggregates problems from AIME (1984-2023), AMC¹ (prior to 2023), Omni-MATH [Gao+24], and the STILL dataset [RUC25], offering strong coverage of competition-level mathematics. Further, DeepScaler has been meticulously pre-filtered to focus exclusively on mathematics with redundant questions removed.

The competition-level problems in DeepScaler span an appropriate difficulty range for surfacing robust reasoning capabilities across diverse mathematical domains. This breadth exposed our generation model to varied problem-solving scenarios, each potentially eliciting different tool-integrated solution approaches, helping create a dataset that could teach effective tool integration.

¹The American Mathematics Competitions (AMC) are a series of examinations and curriculum materials that build problem-solving skills and mathematical knowledge in middle and high school students. The AMC 8, AMC 10, and AMC 12 serve as qualifying exams for the AIME.

3.3.2. LIMO: Extreme Challenge as Quality Signal

While DeepScaler provided breadth, LIMO offered 817 problems of exceptional difficulty, carefully curated from millions of candidates. These problems were selected specifically because they challenge state-of-the-art models like DeepSeek-R1. Our rationale was straightforward: if we could generate successful tool-integrated solutions for these extremely difficult problems, those traces would be particularly valuable training examples.

Successfully solving hard problems with TIR typically requires sophisticated computational strategies, systematic exploration, and careful verification. The traces that emerged from LIMO problems, when successful, tended to demonstrate advanced tool usage patterns. While we expected lower success rates on LIMO compared to DeepScaler, the successful traces would demonstrate tool-integrated reasoning on very difficult problems.

3.4. Prompt Engineering for TIR

The effectiveness of our bottom-up generation approach required careful prompt engineering to guide Gemini 2.0 Flash toward genuine tool-integrated reasoning. Without explicit guidance, the model defaulted to pure mathematical reasoning, avoiding code or using it only superficially for verification. This section describes how we developed prompts that successfully elicited tool-integrated reasoning patterns.

The impact of effective prompt engineering proved substantial. On a representative sample of AIME 2024 problems (with 10x oversampling), a baseline prompt achieved approximately 37% accuracy while using only natural language reasoning. With our carefully engineered tool-integrated reasoning prompt, the same model achieved approximately 57% accuracy, a relative improvement of over 50%. This significant gain came not from enhancing the model’s mathematical capabilities but from guiding it to leverage computational tools as an integral part of its reasoning process.

Table 3.2.: Performance comparison of Gemini 2.0 Flash with and without the code-first TIR prompt on AIME 2024 problems.

Approach	Accuracy	Relative Improvement
Baseline prompt	36.67% (110/300)	—
Code-first TIR prompt	56.67% (170/300)	+54.55%

3.4.1. The Prompt Development Process

Our journey toward an effective TIR prompt began with naive attempts that simply asked models to “use Python code to help solve this problem.” These initial prompts produced disappointing results: models would either ignore the instruction entirely, reverting to pure mathematical reasoning, or use code merely as an afterthought to verify manually derived answers. The core issue was that models had been trained extensively on mathematical solutions that prioritize elegant analytical derivations, creating a strong prior against computational approaches.

To understand why our prompts failed and how to improve them, we developed a systematic analysis pipeline. This pipeline paired each failed solution attempt with the correct answer and asked a language model to analyze where the reasoning went wrong. Crucially, we focused not just on mathematical errors but on tool usage patterns: did the model properly read and interpret code outputs? Did it use computational exploration to guide its reasoning, or merely to confirm preconceived notions?

The analysis revealed several recurring failure modes. Models frequently ignored code outputs that contradicted their analytical reasoning, continuing with flawed approaches despite computational evidence to the contrary. They would generate code that explored only narrow aspects of problems rather than using computation to build comprehensive understanding. Most problematically, they treated code execution as a separate activity from reasoning, creating artificial boundaries between “thinking” and “computing” phases.

Based on these observations, we determined that effective TIR prompts should position code as the primary investigative tool rather than a supplementary confirmation method. Instead of asking models to reason first and compute later, we would instruct them to explore computationally and formalize mathematically only after obtaining computational results. This code-first approach proved more effective at eliciting genuine tool-integrated reasoning.

3.4.2. The Code-First Approach

Our final prompt design inverted the traditional relationship between mathematical reasoning and computation. The prompt, which we call the “Code-First Math Solver” (see Appendix B for the full prompt), explicitly states that “CODE IS REQUIRED & PRIMARY” and instructs models to use code as the primary tool for exploring problems, formulating hypotheses, and deriving intermediate results.

The prompt structures the solution process into distinct phases. After initial problem understanding, the model enters a “CODE EXPLORATION & HYPOTHESIS FORMULATION” phase where it uses code to probe the problem space, test simple cases,

and build intuition. The prompt explicitly instructs: “Do not formulate a complete mathematical derivation before exploring and testing its components with code.”

During solution development, the prompt requires a specific workflow for each step: first formulate a computational task, then implement and execute the code, interpret the results, and only afterwards formalize findings mathematically if necessary. This approach encourages computation to drive the solution rather than serving as an afterthought. The prompt includes domain-specific requirements: geometry problems must use coordinate systems, probability problems require simulation verification, and algebraic problems need numerical testing of symbolic results.

The prompt also treats verification as an integral part of the reasoning process. It requires at least one independent verification method using code and mandates explicit handling of contradictions. If verification reveals any inconsistency, the model must stop immediately, identify the faulty step, and return to earlier stages to revise the approach. This creates a self-correcting loop where computational evidence guides reasoning throughout.

The effectiveness of this approach becomes clear through concrete examples. Consider the octagon coloring problem from AIME, where vertices are randomly colored red or blue and we seek the probability that the octagon can be rotated so all blue vertices land on originally red positions. The baseline approach attempted pure mathematical analysis, incorrectly concluding with a probability of $\frac{35}{128}$ for the final answer. In contrast, the TIR approach immediately began computational exploration, systematically checking all 256 possible colorings and discovering that exactly 115 satisfy the condition, yielding the correct probability $\frac{115}{256}$ and final answer. Figure 3.2 shows an abridged version of this computational approach.

The code-first prompt proved effective across our entire data generation pipeline. By providing models with a concrete framework for integrating computation into reasoning, we improved accuracy and generated training data that exhibits the tool usage patterns needed for fine-tuning. The full prompt, spanning over 200 lines with detailed guidance for different problem domains, verification requirements, and error handling procedures, is included in Appendix B for reference.

This prompt engineering effort was important for our data generation pipeline, as without explicit code-first instructions, Gemini 2.0 Flash generated traces dominated by pure reasoning with superficial tool use. With these instructions, we obtained the volume of examples needed for our filtering pipeline, demonstrating tool-integrated reasoning where computation drives discovery and mathematical formalization documents computational insights. However, this success in generation volume revealed a new challenge: among the traces that achieved correct answers and employed code, quality varied widely in terms of reasoning clarity, code usage patterns, and value as training examples.

Problem Statement

Each vertex of a regular octagon is independently colored either red or blue with equal probability. The probability that the octagon can then be rotated so that all of the blue vertices end up at positions where there were originally red vertices is $\frac{m}{n}$, where m and n are relatively prime positive integers. What is $m + n$?

Assistant

Let me systematically check all possible colorings to find those that satisfy the rotation condition. For each of the 256 possible colorings, I'll check if there exists a rotation where all blue vertices land on red positions.

```
1 def count_favorable_colorings():
2     """Counts the number of colorings that satisfy the condition."""
3     count = 0
4     for i in range(256): # Check all 2^8 possible colorings
5         coloring = np.array([int(bit) for bit in bin(i)[2:].zfill(8)])
6         if check_condition(coloring):
7             count += 1
8     return count
9
10 favorable_count = count_favorable_colorings()
11 print(f"Number of favorable colorings: {favorable_count}")
```

Output

Number of favorable colorings: 115

Assistant

Perfect! The code found 115 favorable colorings out of 256 total possibilities. Therefore, the probability is $\frac{115}{256}$. Since $\gcd(115, 256) = 1$, we have $m = 115$ and $n = 256$.

Final Answer

371

Figure 3.2.: Abridged illustrative tool-integrated solution to the octagon coloring problem using systematic enumeration to avoid analytical counting errors.

3.5. Multi-Stage Quality Control

Generating correct solutions was only the first step. We needed to identify traces that exhibited both structural quality and cognitive behaviors conducive to training effective TIR models.

We developed a two-stage filtering pipeline to address this challenge. The first stage applied heuristic filters for structural quality: balance between natural language and code, appropriate conversation lengths, and clean tool integration patterns. The second stage evaluated cognitive behaviors identified by recent research as essential for self-improving reasoning systems. This approach addressed both surface-level quality and deeper reasoning patterns.

Our filtering approach prioritized quality over quantity, ultimately retaining only 3,410 traces from the 27,826 correct solutions. In the context of tool-integrated reasoning, where each training example must coordinate natural language reasoning with code execution, ensuring quality becomes particularly important.

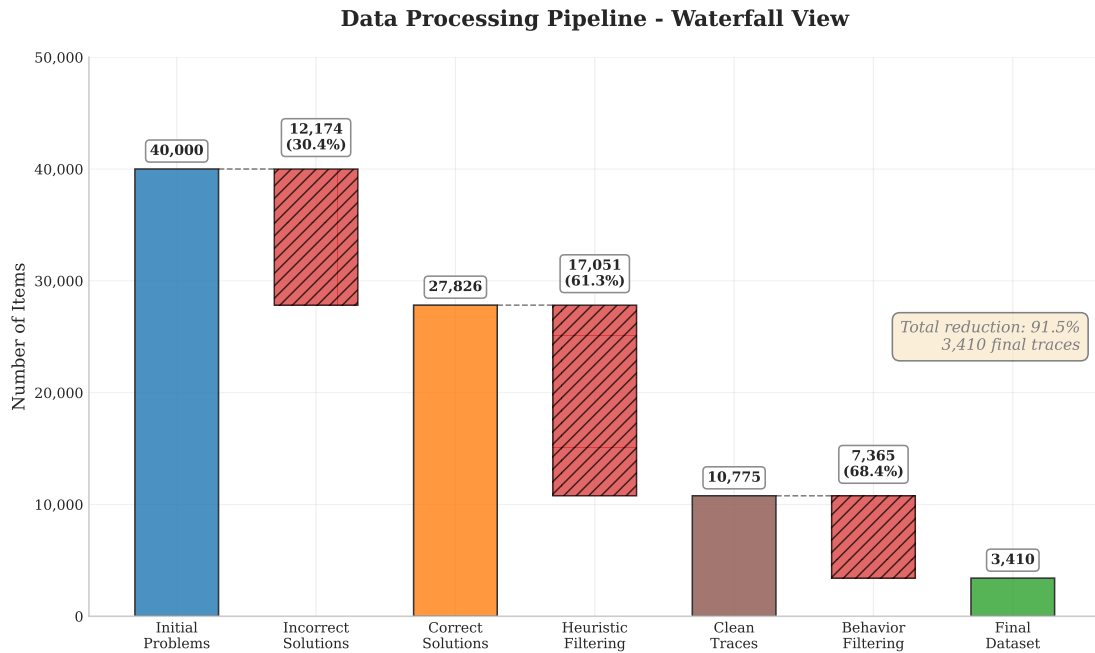


Figure 3.3.: Data reduction pipeline showing the progressive filtering from 40,000 generated traces through correctness verification and quality filtering to 3,410 final training examples. Each stage removes traces based on different criteria, with percentages showing the reduction relative to the previous stage.

3.5.1. Stage 1: Heuristic Filtering

Metric Analysis

Our heuristic filtering stage began with a comprehensive analysis of the generated traces to understand their structural characteristics and identify appropriate quality thresholds. We systematically analyzed the structural characteristics of all generated traces, examining various aspects of conversation structure to identify quality indicators and inform our filtering criteria.

The key metrics we analyzed encompassed both quantitative and qualitative aspects of tool-integrated reasoning traces. Code-to-text ratios, calculated at both character and line levels, revealed the balance between computational exploration and natural language reasoning: traces with too little code often reflected superficial tool use, while those with excessive code lacked the explanatory narrative we sought for training. Conversation length distributions helped identify outliers: extremely short conversations typically indicated trivial problems or premature termination, while excessively long ones often contained repetitive or circular reasoning. Comment density within code blocks provided insight into code clarity; while some comments aid understanding, excessive commenting often indicated degenerate overuse of comments rather than clear, self-explanatory code. We also tracked maximum contiguous non-code sequences, as long stretches of pure text without computational grounding suggested departure from tool-integrated reasoning principles. Finally, we identified code blocks that produced no output, which could indicate errors or visualization attempts.

These metrics were analyzed through comprehensive visualizations that revealed distribution patterns and natural clustering, allowing us to identify sweet spots for effective TIR training examples. Similarly, scatter plots correlating different metrics revealed relationships that informed our multi-criteria filtering approach.

Threshold Determination

The process of determining filtering thresholds combined statistical analysis with manual inspection of edge cases. For each metric, we examined traces at various percentiles to understand what different threshold values would include or exclude. This data-driven approach ensured our thresholds captured meaningful quality distinctions rather than arbitrary cutoffs.

Our final threshold selections reflected a balancing of competing concerns. The conversation length filter of 14,000 to 21,000 characters emerged from analyzing the distribution's core range while excluding outliers. Traces below 14,000 characters often rushed to solutions without adequate exploration or validation, while those exceeding 21,000 characters frequently contained redundant reasoning or unnecessary detours.

The code ratio threshold of 0.35 to 0.70 (measured as assistant code characters divided by total assistant characters) ensured sufficient computational content without overwhelming the natural language narrative. We found that traces below 35% code ratio often used code merely for final confirmation after solving problems analytically, while those above 70% provided insufficient reasoning context. The comment ratio threshold of less than 0.5 (comment lines per code line) filtered out over-commented code. The maximum contiguous non-code threshold of 12 lines prevented long theoretical discussions disconnected from code exploration.

Beyond these numerical thresholds, we also excluded traces with specific problematic patterns. Conversations using `matplotlib` were filtered out because our execution environment could not display visual outputs. Traces containing code that produced no `stdout` or `stderr` often indicated failed code execution or visualization attempts and were likewise discarded.

Filtering Results

Applying our heuristic filters to the dataset revealed the extent of quality variation in the generated traces. Each conversation was evaluated against all criteria, tracking which thresholds were not met.

The impact of heuristic filtering was significant: from the correct traces, we retained 10,775 that met all structural quality criteria, a 61.3% reduction. The filtered dataset showed markedly improved distributions across all metrics, with tighter clustering around optimal values and elimination of problematic outliers.

To validate our filtering decisions, we manually reviewed randomly selected conversations from different regions of the metric space. This review suggested that our thresholds helped distinguish between higher and lower quality TIR examples. Traces passing all filters generally showed improved problem understanding, more systematic computational exploration, better integration of code outputs into reasoning, and more concise solution narratives compared to those that were filtered out.

3. Generating High-Quality TIR Training Data

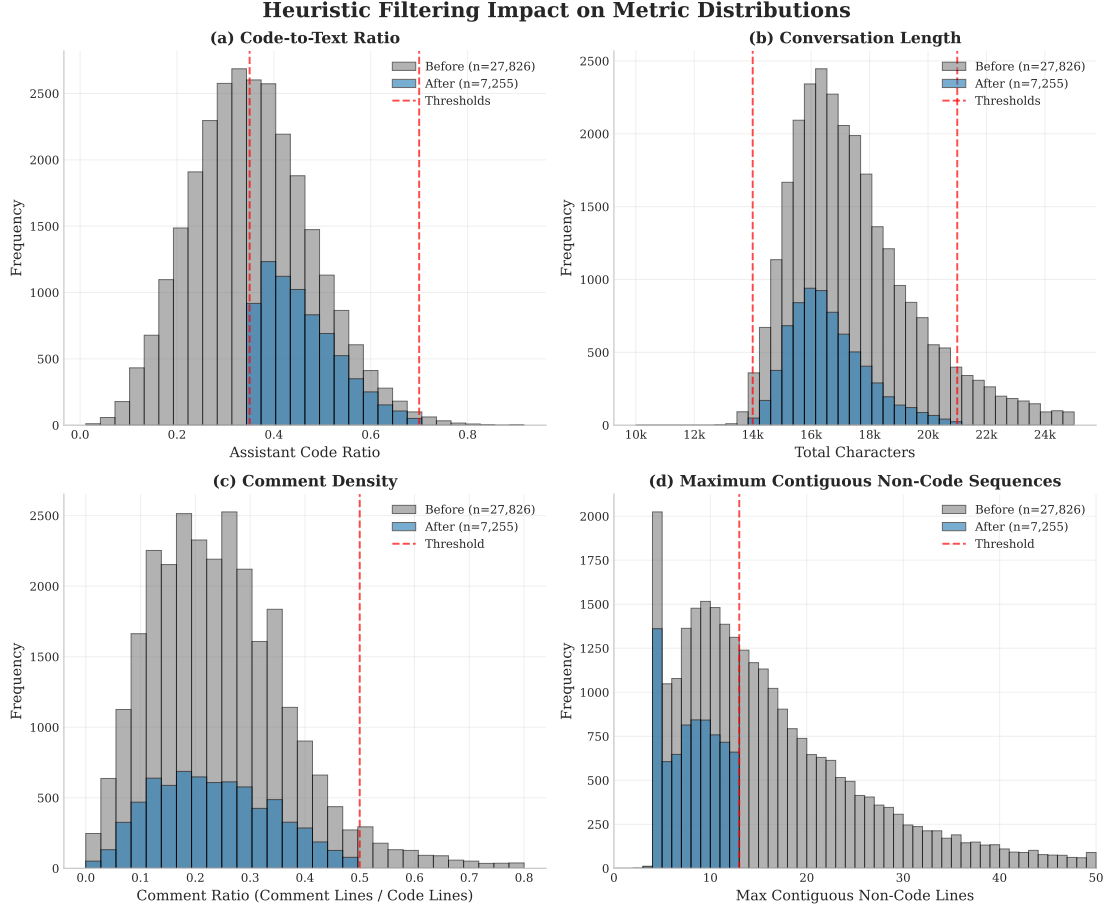


Figure 3.4.: Impact of heuristic filtering on metric distributions. Each panel shows the distribution of a key metric before (gray) and after (blue) filtering, with red dashed lines indicating the filtering thresholds. The significant shifts in distributions demonstrate how filtering selects for traces with balanced code-to-text ratios, appropriate lengths, minimal comment overhead, and continuous computational engagement.

Table 3.3.: Heuristic filtering thresholds applied to ensure structural quality of tool-integrated reasoning traces

Metric	Threshold	Rationale	% Excluded
Length	14–21k chars	Ensures sufficient exploration without excessive verbosity	9.9
Code ratio	0.35–0.70	Balances computational content with natural language narrative	45.3
Comments	< 0.5	Prevents over-commented code that obscures logic	1.6
Non-code	≤ 12 lines	Avoids long theoretical discussions without computation	2.6
Outputs	Required	Ensures genuine code execution	1.7
Libraries	No plot libs	Execution environment cannot display visual outputs	<0.1

3.5.2. Stage 2: Behavior-Based Filtering

Theoretical Foundation

While heuristic filtering ensured structural quality, it could not assess the deeper cognitive patterns that distinguish truly effective reasoning from merely correct solutions. For this second stage of filtering, we turned to recent research on what makes reasoning traces effective for training self-improving models. Gandhi et al. [Gan+25] identified four cognitive behaviors that enable language models to improve through reinforcement learning: verification, backtracking, subgoal setting, and backward chaining. Their crucial finding was that "the presence of reasoning behaviors, rather than correctness of answers, proves to be the critical factor" in enabling self-improvement: a model trained on traces exhibiting these behaviors can learn to reason more effectively, even when some training examples contain errors.

We adopted their framework but recognized that tool-integrated reasoning introduces unique considerations. The four behaviors from Gandhi et al. translate naturally to TIR: verification becomes systematic checking of computational results against problem constraints using explicit validation methods, backtracking involves explicitly revising approaches when code outputs reveal that a solution path proves unsuccessful, subgoal setting means decomposing problems into computational subtasks with clear

intermediate milestones, and backward chaining uses code to work systematically from desired outputs to identify required inputs and prerequisites. However, we identified a fifth behavior specific to tool-integrated reasoning that we termed "Code-Firstness."

Code-Firstness captures whether a model proactively uses code as part of its problem-solving strategy rather than merely for post-hoc verification. A trace scoring high on Code-Firstness begins computational exploration early, uses code to test hypotheses and build understanding, and integrates computational results into subsequent reasoning. In contrast, traces with low Code-Firstness solve problems analytically and only then write code to verify predetermined answers. This distinction proves important for TIR training, as models must learn when and how to use code, not just how to implement predetermined solutions.

Scoring Rubrics and Threshold Justification

Each behavior was scored on a 0-10 scale with detailed rubrics that mapped specific observable patterns to numerical scores. These rubrics are important for interpreting our filtering thresholds, as the numeric scores alone provide little insight into what behaviors we actually selected for. We present here the key distinctions in the scoring rubrics that informed our threshold selection.

Verification

The rubric distinguished between levels of verification:

- **Score 0-2:** No verification or only trivial checks
- **Score 3-4:** Basic verification such as substituting the final answer back into one equation
- **Score 5-6:** Moderate verification with several intermediate steps checked or robust final answer verification
- **Score 7-8:** Strong verification where "most critical intermediate results are checked, and the final solution is verified thoroughly, potentially using a different method or checking edge cases"
- **Score 9-10:** Exceptional systematic verification with multiple methods and validation of critical assumptions

Our threshold of ≥ 8 selects for traces that demonstrate strong verification practices, going beyond simple answer checking to include thorough validation through multiple approaches or edge case testing.

Subgoal Setting

The rubric evaluated problem decomposition:

- **Score 0-2:** No explicit planning or structure, solving proceeds without decomposition
- **Score 3-4:** Basic subgoal setting with simple high-level steps mentioned (e.g., “first find the derivative, then set it to zero”)
- **Score 5-6:** Moderate subgoal setting where “the problem is explicitly broken down into several logical, intermediate steps”
- **Score 7-8:** Strong subgoal setting where “a clear, detailed, and logical plan is outlined early on, breaking the problem into well-defined, manageable subgoals. The solution consistently refers back to or follows this plan”
- **Score 9-10:** Exceptional hierarchical decomposition with sub-subgoals and comprehensive planning

Our threshold of ≥ 7 selects for traces with strong, explicit planning that structures the entire solution process.

Backward Chaining

The rubric assessed goal-directed reasoning:

- **Score 0-2:** No backward reasoning, solution proceeds only forward from givens
- **Score 3-4:** Basic backward chaining with one or two steps working from the goal
- **Score 5-6:** Moderate backward chaining with “several steps of reasoning backward from the goal state”
- **Score 7-8:** Strong backward chaining where “a significant portion of the reasoning works backward from the goal. The structure of the solution is clearly driven by identifying necessary prerequisites derived from the target outcome”
- **Score 9-10:** Exceptional backward chaining that structures the entire solution approach

Our threshold of ≥ 7 selects for traces that demonstrate strong goal-directed reasoning, not just occasional backward steps.

Code-Firstness

Our custom rubric evaluated the overall integration of code throughout the solution, considering timing, purpose, and centrality:

- **Score 0:** No code used in the solution
- **Score 1-2:** Code used only for final verification of results already derived through text/math reasoning
- **Score 3-4:** Code used for significant calculations, but primarily after the main logic has been laid out in text
- **Score 5-6:** Code used for intermediate steps or complex calculations that are part of the reasoning process, potentially setting up the environment for subsequent code-based solving
- **Score 7-8:** Code used early for significant exploration, simulation, or solving major parts of the problem as part of the initial strategy
- **Score 9-10:** The primary method for solving the problem is code-based, with the core logic implemented and executed in code from early on

Our threshold of ≥ 5 selects for traces where code is used as an integral part of the reasoning process, not merely for post-hoc verification.

Backtracking

We deliberately chose not to set a threshold for this behavior because it is inherently problem-dependent; many problems can be solved correctly on the first attempt without needing to abandon any approaches. Backtracking, as implemented in our rubric, requires explicit acknowledgment of failed approaches and strategic shifts to different solution methods, not merely minor error corrections. Our scoring distinguishes between trivial corrections (scores 1-2), moderate strategic adjustments (scores 5-6), and significant strategic pivots where the solver explicitly identifies critical flaws and switches to distinctly different approaches (scores 7-8). Requiring backtracking would artificially bias our dataset toward problems that happen to elicit mistakes during solving, potentially excluding high-quality traces that succeeded on the first well-planned attempt.

Behavior Scoring Implementation

Scoring 10,775 conversations on five cognitive behaviors required an automated approach that could operate at scale while maintaining consistency. We employed Gemini

2.0 Flash as our scoring model, chosen for its consistency with our data generation pipeline and cost-effectiveness at scale. Gemini 2.0 Flash’s efficiency enabled us to score our entire filtered dataset multiple times during rubric refinement without encountering rate limits or excessive costs.

Our scoring approach evaluated each cognitive behavior independently using carefully crafted prompts containing the full scoring rubrics. For each behavior, we formatted the conversation with clear role markers and submitted it to Gemini 2.0 Flash along with the behavior-specific scoring criteria. To ensure scoring quality and interpretability, we required the model to provide a brief analysis before assigning each numerical score. This approach of requiring reasoning before rating helped prevent arbitrary scoring and provided valuable insights into edge cases. The justifications also enabled iterative refinement of our rubrics when we observed systematic scoring patterns that suggested ambiguities in our criteria.

Filtering Results

Determining thresholds for behavior-based filtering required balancing dataset quality with size. Setting thresholds too high would yield a tiny dataset of exemplary traces, while setting them too low would retain traces lacking the cognitive behaviors we sought to instill. We analyzed score distributions for each behavior and selected thresholds that captured traces demonstrating clear presence of desired behaviors while maintaining sufficient data for effective training.

The score distributions revealed distinct patterns for each behavior. These patterns reflected genuine quality characteristics but also Gemini 2.0 Flash’s scoring tendencies. Like most language models, Gemini 2.0 Flash has systematic biases such as preferences for certain numerical ranges or reluctance to assign extreme scores. Therefore, absolute score values should be interpreted within the context of this particular model’s scoring behavior rather than as universal quality metrics. Verification showed very high baseline quality, with 81.7% of traces scoring 9/10 and a median of 9.0, indicating that most successful solutions naturally include strong verification practices. In contrast, Code-Firstness exhibited much greater variance (mean 5.12, standard deviation 2.70), with the distribution spanning the full 0-10 range, validating our decision to set a moderate threshold. Backward Chaining demonstrated a bimodal pattern: 40.6% of traces showed no backward reasoning (score 0), while another significant cluster appeared at scores 7-8, suggesting this behavior appears strongly when present but is not universal across all problem types.

The impact of behavior-based filtering was substantial: from 10,775 heuristically filtered traces, only 3,410 met all behavioral criteria, a 68.4% reduction. This filtering successfully altered the dataset’s behavioral profile. Mean Verification scores increased

modestly from 8.83 to 8.88, as most traces already showed strong verification. Subgoal Setting scores improved from 7.70 to 8.10, indicating clearer problem decomposition in retained traces. The most significant improvement came in Backward Chaining, where mean scores jumped from 6.13 to 7.72. Code-Firstness scores also significantly increased from 5.12 to 6.85, ensuring the final dataset contained traces that genuinely integrated the use of code throughout the reasoning process.

Table 3.4.: Behavior filtering criteria and their impact on dataset composition. All thresholds must be met simultaneously for trace retention.

Behavior	Threshold	Mean (Pre)	Mean (Post)	Rationale
Verification	≥ 8	8.83	8.88	Strong verification: critical results checked thoroughly using different methods or edge case testing
Subgoal Setting	≥ 7	7.70	8.10	Clear, detailed plan outlined early that breaks problem into well-defined subgoals and guides execution
Backward Chaining	≥ 7	6.13	7.72	Significant portion of reasoning works backward from goal, identifying necessary prerequisites
Code-Firstness	≥ 5	5.12	6.85	Code used as integral part of reasoning process throughout solution, not merely for post-hoc verification
Backtracking	None	2.19	2.74	Problem-dependent behavior; not required for all solutions

3.6. Data Generation Results

The complete data generation pipeline, encompassing problem selection, bottom-up generation, prompt engineering, correctness verification, and two-stage quality filtering, yielded our final training dataset.

The final dataset contains 3,410 tool-integrated reasoning conversations, split into 3,069 training examples and 341 validation examples following a 90/10 division. These conversations average 4,747 characters in length, placing them comfortably within the range that allows complete reasoning without excessive verbosity. Conversations

average 2.84 code blocks, with a distribution peak at 3 blocks (48.1% of conversations), indicating consistent code use. The code-to-text ratio of 40.7% represents a balance we found promising through experimentation: enough code to drive the solution computationally while maintaining sufficient natural language narrative to explain the reasoning process.

Quality validation of the filtered dataset included manual review of randomly sampled conversations. This review suggested that our multi-stage filtering successfully identified traces that exemplify effective tool-integrated reasoning.

To illustrate the quality of our final dataset, consider the following example that achieved high scores across all cognitive behaviors:

3.6.1. TIR Example: Music Program Arrangement with Error Recovery

Problem Statement

A class prepared 5 programs to participate in the Xiamen No.1 Middle School Music Square event (this event only has 5 programs), and the order of the programs has the following requirements: Program A must be in the first two positions, Program B cannot be in the first position, and Program C must be in the last position. How many possible arrangements of the program order are there for this event?

Assistant

The problem asks for the number of possible arrangements of 5 programs (A, B, C, D, E) with the following constraints: 1. Program A must be in the first or second position. 2. Program B cannot be in the first position. 3. Program C must be in the last position.

This is a permutation problem with constraints.

Let's start by exploring the possible arrangements using code. We can generate all permutations and then filter them based on the given constraints.

```
1 import itertools
2
3 # Generate all permutations of the programs A, B, C, D, E
4 programs = ['A', 'B', 'C', 'D', 'E']
5 all_permutations = list(itertools.permutations(programs))
6
7 # Function to check if a permutation satisfies the constraints
8 def check_constraints(permutation):
9     # Program A must be in the first two positions
10    if permutation[0] != 'A' and permutation[1] != 'A':
11        return False
12    # Program B cannot be in the first position
13    if permutation[0] == 'B':
14        return False
15    # Program C must be in the last position
16    if permutation[-1] != 'C':
17        return False
18    return True
19
20 # Filter the permutations based on the constraints
21 valid_permutations = [p for p in all_permutations if
22                        check_constraints(p)]
23
24 # Print the number of valid permutations
25 print(f"Total number of permutations: {len(all_permutations)}")
26 print(f"Number of valid permutations: {len(valid_permutations)}")
27
28 # Print some valid permutations to verify
29 print("Some valid permutations:")
30 for i in range(min(5, len(valid_permutations))):
31     print(valid_permutations[i])
```

Output

```
Total number of permutations: 120
Number of valid permutations: 10
Some valid permutations:
```

```
('A', 'B', 'D', 'E', 'C')
('A', 'B', 'E', 'D', 'C')
('A', 'D', 'B', 'E', 'C')
('A', 'D', 'E', 'B', 'C')
('A', 'E', 'B', 'D', 'C')
```

Assistant

The code exploration shows that there are 10 valid permutations. Let's analyze this result and try to derive it mathematically.

We have the following constraints: 1. A is in the first or second position. 2. B is not in the first position. 3. C is in the last position.

Let's consider two cases:

Case 1: A is in the first position. - The arrangement is A _ _ C. - B cannot be in the first position, which is already satisfied. - The remaining positions can be filled by B, D, and E in any order. - There are $3! = 6$ ways to arrange B, D, and E.

Case 2: A is in the second position. - The arrangement is _ A _ C. - B cannot be in the first position. - The first position can be filled by either D or E. - If the first position is D, the arrangement is D A _ C. The remaining positions can be filled by B and E in any order, so there are $2! = 2$ ways. - If the first position is E, the arrangement is E A _ C. The remaining positions can be filled by B and D in any order, so there are $2! = 2$ ways. - So, there are $2 + 2 = 4$ ways in this case.

Therefore, the total number of possible arrangements is $6 + 4 = 10$.

Let's verify the result using a different approach. We can use SymPy to generate all permutations and filter them based on the constraints. This will provide an independent verification of our result.

```
1 import sympy
2 from sympy.combinatorics import Permutations
3
4 # Define the programs
5 programs = ['A', 'B', 'C', 'D', 'E']
6
7 # Generate all permutations
8 all_permutations = list(Permutations(programs).list())
9
```

```
10 # Function to check if a permutation satisfies the constraints
11 def check_constraints(permutation):
12     # Program A must be in the first two positions
13     if permutation[0] != 'A' and permutation[1] != 'A':
14         return False
15     # Program B cannot be in the first position
16     if permutation[0] == 'B':
17         return False
18     # Program C must be in the last position
19     if permutation[-1] != 'C':
20         return False
21     return True
22
23 # Filter the permutations based on the constraints
24 valid_permutations = [p for p in all_permutations if
25                        check_constraints(p)]
26
27 # Print the number of valid permutations
28 print(f"Number of valid permutations (SymPy): {len(valid_permutations)}")
```

Error

```
File "<string>", line 2, in <module>
ImportError: cannot import name 'Permutations' from 'sympy.combinatorics'
(Users/christo/projects/arc-agi-solver/.venv/lib/python3.11/site-
packages/sympy/combinatorics/__init__.py)
```

Assistant

The 'Permutations' class is deprecated in recent versions of SymPy. We should use 'itertools' instead, as we did in the initial exploration. The error indicates that the verification code is not independent. Let's modify the verification code to use a different approach. We will generate all possible arrangements based on the constraints and then count them.

```
1 # Verification using a different approach: generating arrangements
   based on constraints
2
3 def generate_arrangements():
4     arrangements = []
5     # Case 1: A is in the first position
6     for p in itertools.permutations(['B', 'D', 'E']):
7         arrangements.append(('A', p[0], p[1], p[2], 'C'))
8
9     # Case 2: A is in the second position
10    for first in ['D', 'E']:
11        remaining = ['B'] + [x for x in ['D', 'E'] if x != first]
12        for p in itertools.permutations(remaining):
13            arrangements.append((first, 'A', p[0], p[1], 'C'))
14    return arrangements
15
16 arrangements = generate_arrangements()
17 print(f"Number of valid arrangements (generated): {len(arrangements)}")
18
19 # Convert to tuples for comparison
20 arrangements_set = set(tuple(x) for x in arrangements)
21 valid_permutations_set = set(valid_permutations)
22
23 # Check if the generated arrangements are the same as the filtered
   permutations
24 if arrangements_set == valid_permutations_set:
25     print("The generated arrangements match the filtered permutations.")
26 else:
27     print("The generated arrangements do NOT match the filtered
   permutations.")
28     print("Generated arrangements:", arrangements_set)
29     print("Filtered permutations:", valid_permutations_set)
```


Output

Number of valid arrangements (generated): 10
The generated arrangements match the filtered permutations.

Assistant

The verification confirms that there are 10 valid arrangements.
By exploring the problem with code, formulating a mathematical approach, and verifying the result with an independent method, we have found that there are 10 possible arrangements of the program order that satisfy the given constraints.
Therefore, the final answer is

Final Answer

10

This example showcases several important patterns that appear throughout our dataset: First, it demonstrates natural error recovery when the model encounters an `ImportError` with a deprecated SymPy feature, adapting seamlessly by implementing an alternative verification approach. Second, it shows systematic exploration through code with three distinct code blocks that progressively build the solution. Third, it exhibits clear problem decomposition, analyzing constraints before implementation. Finally, it performs thorough verification by generating arrangements through two independent methods and confirming their equivalence. The trace exemplifies how our filtering pipeline successfully identified examples where models handle real-world programming challenges while maintaining rigorous mathematical reasoning.

Table 3.5.: Final dataset composition after multi-stage quality filtering

Dataset Characteristic	Value
Total conversations	3,410
Training set	3,069 (90%)
Validation set	341 (10%)
Average conversation length	4,747 characters
Average turns per conversation	7.68
Average code blocks per trace	2.84
Code-to-text ratio	40.7%
Code block distribution	
1 code block	41 (1.3%)
2 code blocks	1,026 (33.4%)
3 code blocks	1,476 (48.1%)
4 code blocks	401 (13.1%)
5 code blocks	125 (4.1%)

Our data generation effort demonstrates that systematic quality filtering enables the creation of high-quality training datasets for tool-integrated reasoning. By progressively filtering based on both structural metrics and cognitive behaviors, we created a dataset where the examples demonstrate the reasoning patterns we want models to learn.

3.7. Summary

This chapter traced our journey to generate high-quality tool-integrated reasoning training data through a multi-stage pipeline. We began with a failed attempt to transform existing pure mathematical solutions, discovering that authentic code execution outputs cannot reliably be predicted or simulated. This led us to develop a bottom-up generation approach using Gemini 2.0 Flash. We explored how careful prompt engineering improved performance from 36.67% to 56.67% on AIME 2024, developing a code-first approach. Our two-stage filtering pipeline applied both heuristic metrics and cognitive behavior assessment inspired by recent research. Our pipeline reduced approximately 40,000 generated traces to 3,410 high-quality examples.

4. Building the TIR System

Having developed a multi-stage data generation pipeline in Chapter 3 as our proposed approach to Research Question 1, we now validate its effectiveness by demonstrating how it enables tool-integrated reasoning capabilities. This chapter details the process of transforming a general-purpose language model into a capable TIR agent. Our approach encompasses model selection, system architecture design, experimental optimization, and practical deployment considerations.

We demonstrate that our data generation approach from Chapter 3 successfully transforms the base Qwen3-8B model, which does not exhibit tool-integrated reasoning behavior despite explicit instruction, into a capable TIR agent achieving 28.33% accuracy on AIME 2024.

4.1. Model Selection and Adaptation

4.1.1. Base Model Selection

Building an effective TIR model begins with selecting a pre-trained language model suitable for fine-tuning on tool-integrated reasoning data. This choice involves trade-offs between the computational cost of inference, demonstrated reasoning capabilities pre-fine-tuning, and amenability to fine-tuning for tool use.

The 8B Parameter Sweet Spot

We selected the 8-billion parameter scale as a promising balance between capability and computational tractability. This choice reflects established practices in the LLM community, where models in the 7B-8B range have emerged as a promising trade-off point for much experimental research.

The theoretical foundation for this choice comes from scaling law research [Kap+20], which demonstrates that performance improvements follow power-law scaling with diminishing returns as model size increases. While the paper finds consistent scaling across orders of magnitude, the 8B scale represents a practical balance point where diminishing returns become more significant relative to computational costs, making it particularly attractive for academic research settings.

Qwen3-8B Selection Rationale

Among available 8B-scale models, Qwen3-8B emerged as a promising choice based on comprehensive benchmark performance and architectural considerations. According to the Qwen3 Technical Report [Tea25], Qwen3-8B-Base significantly outperforms comparable models across mathematical reasoning benchmarks.

The model demonstrates particularly strong performance on mathematical reasoning benchmarks, significantly outperforming comparable models like Llama-3-8B [Gra+24]. This mathematical reasoning superiority aligned well with our tool-integrated reasoning objectives, where systematic problem-solving and verification skills may transfer directly.

More importantly, Qwen3 incorporates specific architectural enhancements for reasoning tasks, including enhanced pre-training on STEM data and integration of both “thinking” and “non-thinking” reasoning modes for complex multi-step problems. These design choices suggested the model might be particularly amenable to fine-tuning for tool-integrated reasoning applications.

4.1.2. LoRA Configuration

We employed Low-Rank Adaptation (LoRA) [Hu+21] for fine-tuning, a choice driven by both computational considerations and theoretical advantages for our use case.

LoRA reduces trainable parameters by 10,000x and enables training on constrained hardware with 3x memory reduction. While it generally underperforms full fine-tuning, this trade-off was acceptable for our research objectives.

The QLoRA paper [Det+23] provides useful insights for rank selection. Their experiments demonstrated that when LoRA is applied to all linear transformer layers, rank values from 8 to 256 show minimal performance differences, provided the rank exceeds a minimum threshold.

Our final LoRA configuration employed rank=64 and alpha=128, representing a 1:2 rank-to-alpha ratio. This selection was informed by recent best practices.

4.2. System Architecture and TIR Engine

The TIR system orchestrates integration between natural language reasoning and computational tool usage. Our implementation builds upon established patterns for multi-turn conversation management.

4.2.1. Multi-Turn TIR Orchestration

The core TIR implementation centers on a multi-turn conversation loop that alternates between natural language reasoning and Python code execution, maintaining persistent state across all interactions.

The orchestration process follows this pattern:

1. **Initialization:** Create conversation with system message and problem statement
2. **Model interaction:** Generate response through language model API calls
3. **Content parsing:** Extract and validate natural language reasoning and code blocks
4. **Code execution:** Run Python code in isolated environment with state persistence
5. **Result integration:** Append execution outputs back to conversation history
6. **Termination checking:** Evaluate for final answer, maximum turns, or error conditions

This architecture creates a computational dialogue where insights from code execution inform subsequent reasoning steps, and natural language reasoning guides computational exploration. The persistent conversation state ensures that variables, imported libraries, and computational results remain available throughout the entire problem-solving process.

Figure 4.1 visualizes this orchestration process as a state machine, illustrating the cyclic nature of the conversation loop and the decision points that guide execution flow.

4.2.2. Code Execution Framework

The code execution component employs a code executor that provides a Python environment with mathematical libraries pre-imported.

Execution Environment Design

The execution environment includes several key features:

- **Pre-imported libraries:** NumPy and SymPy are automatically available, reducing overhead and enabling immediate mathematical computation
- **State persistence:** Variables and function definitions persist across code blocks within a single problem-solving session

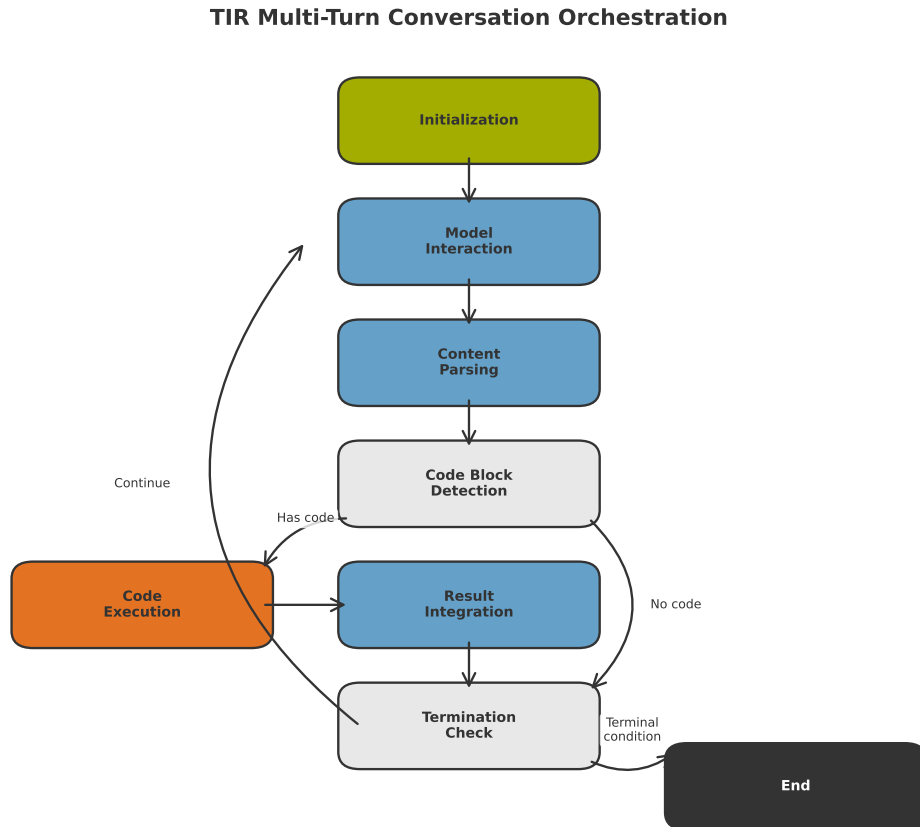


Figure 4.1.: State diagram of the TIR multi-turn conversation orchestration process. The system alternates between natural language reasoning and code execution in a persistent computational dialogue.

- **Output capture:** Both stdout and stderr streams are captured and integrated into the conversation flow
- **Timeout protection:** Code execution includes 4-second timeouts to prevent run-away computations
- **Error handling:** Execution failures are gracefully handled and reported back to the conversation

Integration with Conversation Flow

Code execution results are integrated into the conversation through structured output formatting. When code blocks are detected in model responses, they are extracted, executed, and their outputs appended to the conversation as user messages. This creates a natural dialogue flow where the model can observe and respond to the actual results of its code.

4.2.3. Answer Verification System

Mathematical problem solving requires robust verification mechanisms to assess solution correctness across diverse answer formats. We use Math-Verify¹ for answer verification, which provides multi-tier parsing and comparison capabilities specifically designed for mathematical expressions.

Math-Verify supports multiple extraction configurations to handle diverse answer formats:

- **LaTeX extraction:** Processes mathematical expressions in LaTeX format (e.g., $\frac{1}{2}$)
- **Expression extraction:** Handles plain mathematical expressions (e.g., 1/2)
- **String extraction:** Processes string literals and simple numerical answers

The system attempts to parse both the model’s generated answer and the expected solution in each format, enabling flexible matching across different representation styles. Math-Verify also provides advanced parsing capabilities for set theory, equations, and complex mathematical representations, making it well-suited for the diverse answer types encountered in competition mathematics.

¹Math-Verify: A mathematical expression evaluation system for LLMs. Available at: <https://github.com/huggingface/Math-Verify>

4.3. Fine-Tuning Experiments

We conducted numerous fine-tuning experiments, systematically exploring hyperparameter configurations to identify effective parameters for fine-tuning on TIR data.

4.3.1. Hyperparameter Experimentation

The experimental program was designed as a systematic exploration of the hyperparameter space, progressing from broad parameter sweeps to focused optimization of promising configurations. All experiments were conducted on consistent hardware (4 A100 GPUs) with standardized training protocols to ensure fair comparison across configurations.

Model Evolution and Selection

Our experimental approach evolved through three distinct phases, each reflecting growing understanding of effective TIR model development:

1. **Phase 1:** Initial exploration focused on Qwen/Qwen2.5-Math-7B-Instruct (31 experiments), the math-specialized variant that initially seemed promising for mathematical reasoning tasks.
2. **Phase 2:** Diversification to include Qwen/Qwen2.5-Coder-7B-Instruct (5 experiments) and general-purpose Qwen/Qwen2.5-7B-Instruct (3 experiments) to test the impact of different specializations.
3. **Phase 3:** Transition to Qwen/Qwen3-8B (6 experiments), the newest architecture that ultimately proved most performant.

This progression demonstrates the value of systematic exploration: the math-specialized model that initially seemed optimal was ultimately outperformed by the general-purpose Qwen3-8B architecture, confirming that general-purpose models can excel at specialized tasks when fine-tuned.

Table 4.1 summarizes the distribution of experiments across different base models.

Hyperparameter Space Exploration

The experiments systematically explored key hyperparameter dimensions:

LoRA Configuration: Rank values from 16 to 128 were tested, with rank=64 emerging as the most frequent choice (30 experiments). Alpha values ranged from 8 to 256,

Table 4.1.: Base Model Distribution

Base Model	Experiments	Rationale
Qwen/Qwen2.5-Math-7B-Instruct	31	Math-specialized foundation
Qwen/Qwen3-8B	6	Newer architecture, final choice
Qwen/Qwen2.5-Coder-7B-Instruct	5	Code specialization for TIR
Qwen/Qwen2.5-7B-Instruct	3	Baseline comparison
google/gemma-3-1b-it	2	Size scaling exploration

with the traditional 2:1 rank-to-alpha ratio (e.g., rank=64, alpha=32) appearing in 15 experiments, though the top performers ultimately used 1:2 ratios.

Learning Rate Optimization: Learning rates from $5e-5$ to $4e-4$ were evaluated, with $1e-4$ proving most stable across different configurations (26 experiments). Higher learning rates ($4e-4$) generally led to training instability, while lower rates ($5e-5$) showed slower convergence without clear performance benefits.

Training Duration: Epochs ranged from 1.0 to 12.0, with most experiments using 3.0 epochs (23 experiments). The best-performing training duration varied significantly across different model architectures and hyperparameter configurations, highlighting the importance of checkpoint-based evaluation for identifying peak performance.

Performance Variance and Evaluation Challenges

Evaluation on the AIME 2024 benchmark without oversampling revealed significant performance variance across experiments, even those with similar configurations. This variance stems from several factors:

- **Small evaluation set:** AIME 2024 contains only 30 problems, making performance sensitive to individual problem successes
- **Generation stochasticity:** Even with fixed temperature settings, model outputs contain inherent randomness
- **Task difficulty:** Competition-level mathematics problems create evaluation scenarios where small configuration changes can have disproportionate impact

These observations underscore the importance of oversampling or multiple evaluation runs and careful statistical analysis when working with challenging mathematical reasoning benchmarks.

4.3.2. Final Configuration Selection

Based on the comprehensive experimental exploration, we selected the configuration that achieved the best performance on our primary evaluation metric: AIME 2024 accuracy with 8x oversampling.

Table 4.2 shows the top-performing experimental configurations on AIME 2024.

Table 4.2.: Top 5 AIME 2024 Performing Configurations

Rank	Base Model	LoRA Config	Learning Rate	Epochs	AIME %
1	Qwen3-8B	64/128	1e-4	10	28.33
2	Qwen2.5-Math-7B	64/128	2e-4	2	25.98
3	Qwen3-8B	64/128	1e-4	3	22.66
4	Qwen2.5-Math-7B	64/32	1e-4	5	20.00
5	Qwen2.5-7B-Instruct	64/128	1e-4	3	18.75

Winning Configuration

The selected configuration employed the following settings:

- **Base model:** Qwen/Qwen3-8B
- **LoRA configuration:** rank=64, alpha=128 (1:2 ratio)
- **Learning rate:** 1e-4 with AdamW optimizer
- **Training duration:** 10 epochs total
- **Optimal checkpoint:** step 1536 (8.0 epochs)
- **Final performance:** 28.33% on AIME 2024 (8x oversampling)

Training Dynamics

Analysis of the training curves reveals healthy learning dynamics throughout the optimization process, as demonstrated in Figure 4.2. The training loss exhibits consistent reduction from an initial value of 0.599 to a final value of approximately 0.090, representing an 85% reduction that indicates effective learning of the TIR task structure. The gradient norm evolution demonstrates stable optimization throughout training, with values remaining within a healthy range that indicates neither vanishing nor exploding gradients.

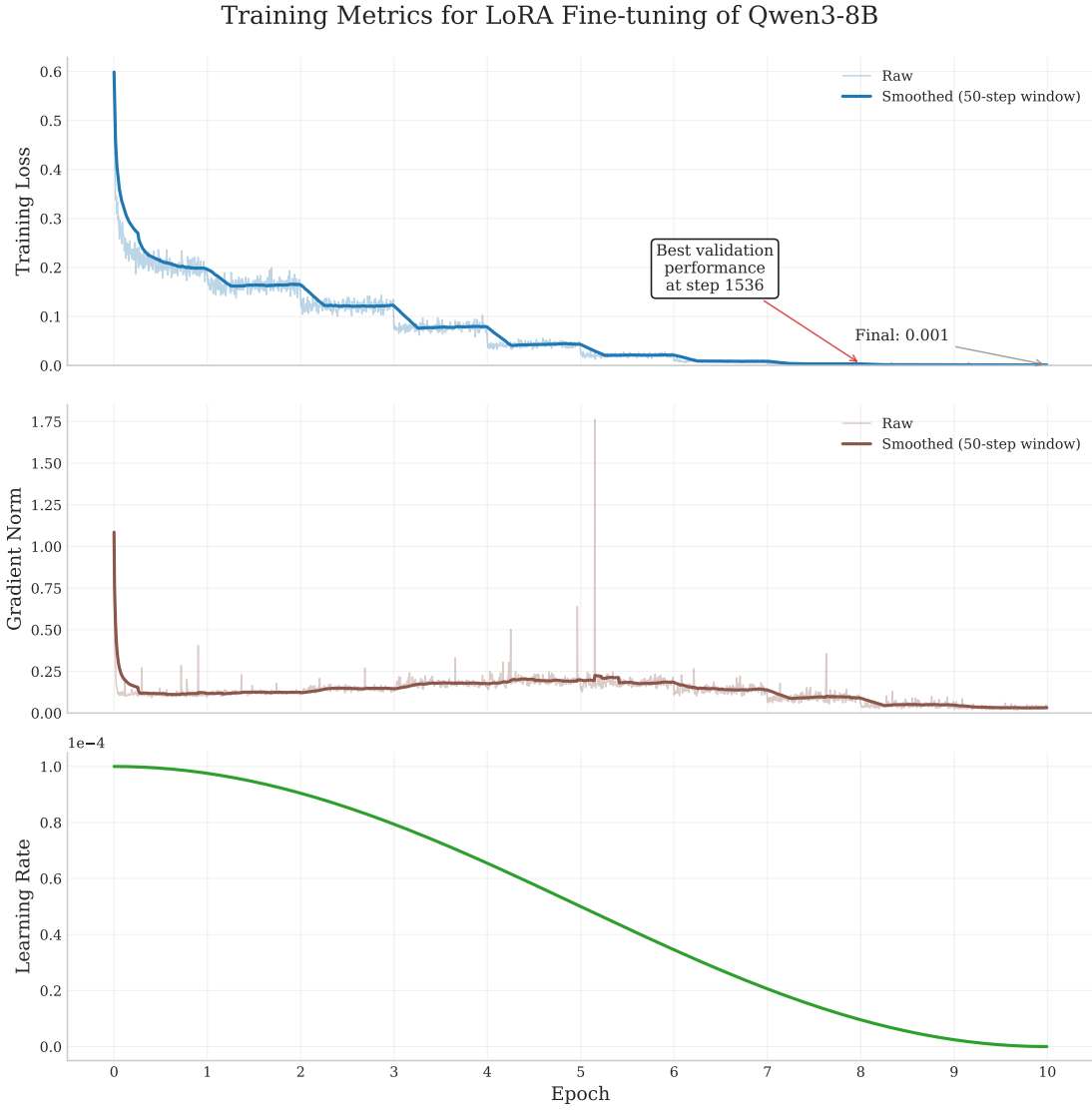


Figure 4.2.: Training dynamics for LoRA fine-tuning of Qwen3-8B showing (top) training loss with best validation checkpoint marked at step 1536, (middle) gradient norm evolution demonstrating stable optimization, and (bottom) learning rate schedule over 10 epochs. The consistent loss reduction and stable gradient norms indicate healthy training dynamics without optimization difficulties.

The identification of the best checkpoint at this point suggests that the model successfully acquired the essential TIR patterns while maintaining good generalization performance.

4.4. Deployment Optimization

We employed Activation-aware Weight Quantization (AWQ) to reduce model size while maintaining acceptable performance levels for deployment.

4.4.1. AWQ Quantization Strategy

We employed AWQ [Lin+24] for model quantization, a choice motivated by its superior balance of performance retention and efficiency gains compared to alternative quantization approaches.

Configuration Rationale

Our AWQ configuration was informed by empirical research validating specific parameter choices:

4-bit Quantization: We selected 4-bit precision based on comprehensive research by Dettmers et al. [DZ22], whose study of over 35,000 zero-shot experiments concluded that “4-bit precision is almost universally optimal for total model bits and zero-shot accuracy.”

Group Size 128: This represents the standard configuration used throughout the AWQ paper. Group size 128 provides a balance between quantization granularity and overhead, as demonstrated across experiments on models ranging from 7B to 70B parameters.

128 Calibration Samples: This sample size has been validated across multiple quantization studies. Frantar et al. [Fra+23] successfully quantized models up to 175B parameters using only 128 samples, demonstrating efficient quantization within approximately 4 GPU hours. Our calibration dataset consisted of 128 TIR conversations sampled from our training data, ensuring representative coverage of the target task distribution.

4.4.2. Performance Trade-off Analysis

Quantization inevitably involves trade-offs between model size, inference efficiency, and task performance. Our evaluation methodology assessed these trade-offs using the same AIME 2024 benchmark employed for model selection.

Quantization Impact Assessment

Table 4.3 summarizes the performance trade-offs achieved through AWQ quantization on AIME 2024 evaluation with 8x oversampling.

Table 4.3.: AWQ Quantization Performance Analysis

Model Version	Accuracy	Performance Loss	Size Reduction	Speed
Original (FP16)	28.33%	– (baseline)	–	–
AWQ 4-bit	25.83%	-2.5pp	~75%	3x

The results demonstrate that AWQ quantization preserved the model’s reasoning capabilities to an acceptable degree while delivering significant practical benefits. The 2.5 percentage point decrease (8.8% relative) represents an acceptable trade-off given the substantial computational advantages achieved.

The quantization process delivered improvements across multiple dimensions: model size decreased by approximately 75% (from ~16GB to ~4GB), inference speed increased by 3x compared to the FP16 baseline, and memory requirements were substantially reduced to enable multi-GPU deployment.

These efficiency gains proved valuable for generating the thousands of rollouts needed to create training data for PRMs. In this context, the performance trade-off was acceptable since the focus shifts from achieving maximum accuracy on individual problems to generating diverse reasoning trajectories at scale within computational budgets.

4.5. Summary

This chapter detailed the systematic construction of our tool-integrated reasoning system, from model selection through deployment optimization. We explored the experiments and practical considerations that led to selecting Qwen3-8B as our base model, finding that this general-purpose model outperformed older math-specialized variants when fine-tuned for TIR. Through numerous fine-tuning experiments, we systematically explored the hyperparameter space and identified an optimal configuration achieving 28.33% accuracy on AIME 2024. The implementation required careful orchestration of multi-turn conversations, persistent code execution state, and robust answer verification. We concluded with deployment optimization through AWQ quantization, achieving 75% model size reduction with only 2.5 percentage points performance loss.

5. Process Reward Models for TIR

Having successfully built a tool-integrated reasoning system in Chapter 4, we now address a key limitation: single-pass generation without mechanisms to explore alternatives or evaluate partial progress. This chapter presents our adaptation of implicit process reward models to enable efficient test-time search for tool-integrated reasoning.

5.1. The Test-Time Improvement Challenge

The single-pass generation approach becomes particularly limiting when tackling competition-level mathematics problems like those in AIME, where multiple promising solution paths may exist but only certain approaches lead to correct and tractable solutions. Without mechanisms to assess the promise of partial solutions or guide search toward more fruitful trajectories, even capable models can become trapped in dead ends.

This challenge motivates the need for test-time search guidance through Process Reward Models, which evaluate partial reasoning trajectories during the solution process. However, traditional PRM training approaches like Math-Shepherd require high computational overhead compared to standard outcome supervision for generating training data, making them impractical for resource-constrained settings.

The recent development of implicit PRMs by Yuan et al. [Yua+24] addresses this bottleneck. By parameterizing rewards as log-likelihood ratios between policy and reference models, implicit PRMs can be trained using only outcome labels while still providing step-level process rewards at inference time. This approach enables effective test-time search for tool-integrated mathematical reasoning without the prohibitive costs of Monte Carlo methods.

The remainder of this chapter details our adaptation of implicit PRMs to handle the structure of tool-integrated reasoning and describes our scalable pipeline for generating training data.

5.2. Adapting PRMs to TIR

Process reward models evaluate reasoning quality at discrete intervals, making the definition of these evaluation boundaries important for effective search guidance. While we could theoretically evaluate after every token (maximum granularity) or only at trajectory completion (reducing to outcome supervision), practical deployment benefits from meaningful intermediate checkpoints that capture complete units of progress. For tool-integrated reasoning, this raises the question of what constitutes an appropriate "step" given the interleaved nature of natural language and code execution. Traditional mathematical reasoning might define steps at the granularity of individual sentences or equations, which are natural units that align with human problem-solving. Tool-integrated reasoning, however, introduces a more complex structure where reasoning, code generation, and execution outputs form interconnected cycles.

Consider a typical TIR trajectory solving a combinatorial problem:

1. Natural language analysis identifying the problem structure
2. Code implementation of a counting algorithm
3. Execution output revealing intermediate results
4. Reasoning about the output and planning next steps
5. Further code to verify or extend the solution

Evaluating any component in isolation does not adequately assess the complete reasoning process. The natural language might propose a correct approach, but the code could contain implementation errors. Conversely, working code might produce correct outputs that the model misinterprets in subsequent reasoning.

We therefore define a TIR step as a complete tool-use cycle encompassing:

- The natural language reasoning leading to tool use
- The generated code block
- The execution output from that code

This holistic definition ensures that our PRM evaluates the quality of complete reasoning-computation units rather than fragmentary components. A high-quality step successfully translates mathematical insights into computational exploration, executes correctly, and properly integrates results into the ongoing reasoning process. This definition naturally aligns with how our TIR system structures conversations, where each assistant turn typically contains one or more complete tool-use cycles.

5.3. Generating Training Data

As introduced in Section 5.1, implicit PRMs offer significant training data efficiency by requiring only outcome labels rather than expensive step-level annotations. This section describes how we generated diverse sets of tool-integrated reasoning trajectories to provide the contrastive signal necessary for effective PRM training.

We selected the DeepScaler dataset [Luo+25] for PRM rollout generation based on several considerations. First, its difficulty distribution creates a proper contrastive dataset well-suited for PRM training. Problems that are too easy yield mostly successful trajectories (no negative signal), while overly difficult problems yield mostly failures (no positive signal). DeepScaler’s AIME-level problems strike the right balance, ensuring sufficient examples of both successful and unsuccessful reasoning trajectories. Second, the dataset offers excellent topical diversity spanning algebra, geometry, combinatorics, and number theory, ensuring our PRM learns general reasoning evaluation rather than domain-specific heuristics. Third, DeepScaler’s careful filtering excludes problems from standard test sets, preventing data contamination that could inflate our evaluation metrics.

Our generation parameters balanced diversity with computational constraints:

8 attempts per problem: Following the protocol established by Yuan et al. [Yua+24], who demonstrated strong results with 8 rollouts per problem in their implicit PRM work, we generated 8 solution attempts for each problem.

Rollout temperature range 0.25-0.75: We determined this range through preliminary experiments. Temperatures below 0.25 produced unimaginative solutions and little diversity, while temperatures above 0.75 were more prone to computational errors and fanciful reasoning.

Maximum 8 turns per conversation: Most successful solutions in our training data complete within 8 turns. This limit prevents runaway conversations that tend toward circular reasoning while accommodating the typical length distribution of TIR trajectories.

Context window of 12,000 tokens: While our model supports larger contexts, very few conversations in our data exceed this limit. This practical bound ensures efficient processing while maintaining coverage of realistic problem-solving trajectories.

Dataset Scale: Our dataset size evolved through systematic experimentation. We began with 2,000 problems to validate the implicit PRM approach on tool-integrated reasoning, generating 16,000 trajectories (8 attempts \times 2,000 problems). The resulting PRM training loss curves showed clear learning progress and discriminative capability, confirming the viability of our approach. These encouraging results with 2,000 problems suggested that

scaling to 8,000 problems could yield a sufficiently capable PRM, though subsequent evaluation hinted that further scaling might provide additional benefits, a direction we note for future work.

This represents approximately one-quarter of the 33,000 problems used in the original implicit PRM work by Yuan et al. [Yua+24]. Our results confirm that implicit PRMs can provide meaningful guidance even with this reduced dataset, though as noted above, additional data would likely yield further improvements.

5.4. Training the PRM

5.4.1. Training Configuration

Following Yuan et al. [Yua+24], we adopted Llama-3.1-8B-Instruct as our base model for the PRM. This choice aligns with their successful deployment of implicit PRMs at similar scale, demonstrating that 8B-parameter models provide sufficient capacity for learning effective process rewards.

Our PRM training leverages their open-source implementation¹, which provides optimized implicit reward objectives and distributed training infrastructure.

Key training configurations include:

- **Single epoch training:** Following the implicit PRM methodology, we train for one complete pass through the data.
- **Batch size:** 256 trajectories with gradient accumulation to simulate larger batches
- **Learning rate:** 1×10^{-6} with cosine annealing
- **Optimizer:** AdamW with default parameters
- **Distributed training:** 8 nodes with A100 GPUs each using DeepSpeed ZeRO Stage 2

This configuration closely follows the methodology established by Yuan et al. [Yua+24].

5.4.2. Training Dynamics

Figure 5.1 illustrates the training dynamics of our implicit PRM across the single training epoch. The curves show several key patterns that distinguish PRM training from standard fine-tuning:

¹Available at <https://github.com/PRIME-RL/ImplicitPRM>

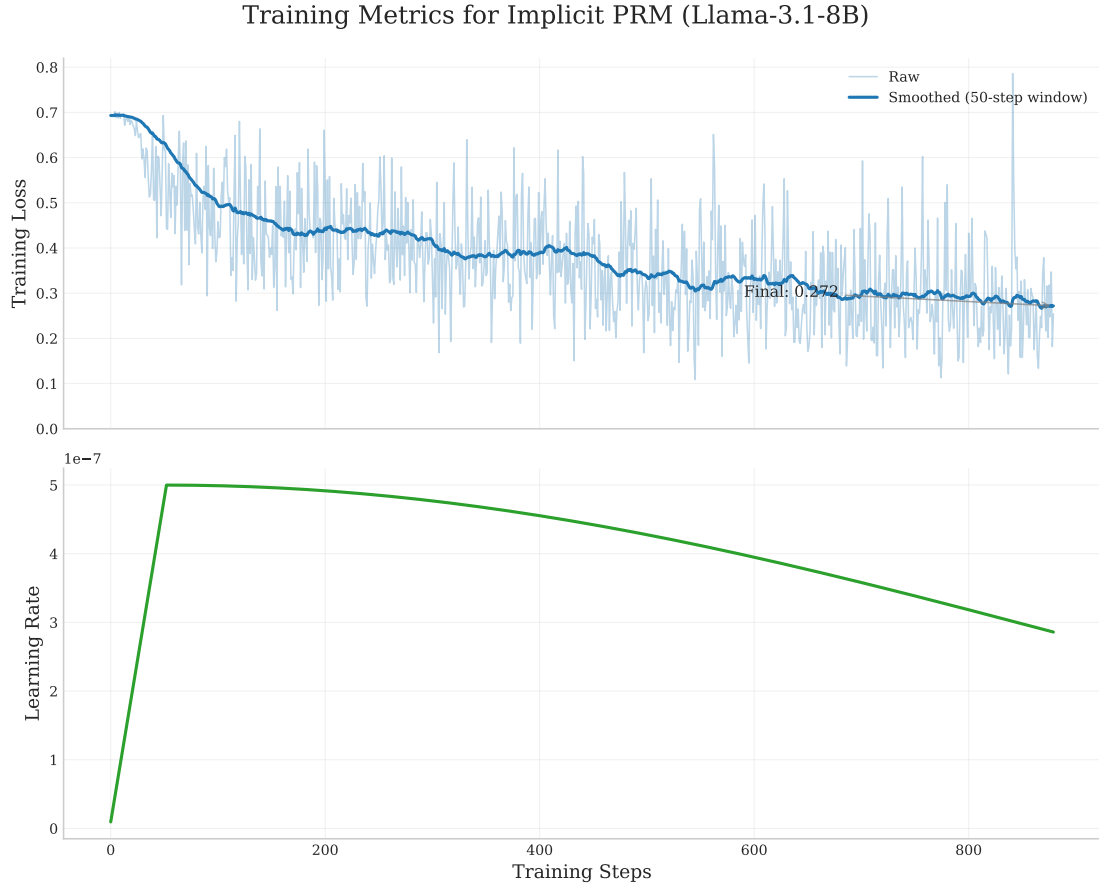


Figure 5.1.: PRM training dynamics showing loss progression and learning rate schedule over the single training epoch. Despite notable variance in the training loss, the overall trend shows steady improvement of the implicit reward model from ~ 0.7 to ~ 0.27 .

Training Variance: The PRM training loss curves exhibit much higher variance compared to the smooth curves observed in TIR model training (Figure 4.2). However, despite this variance, the model demonstrates clear learning progress throughout the epoch.

Learning Progress: The loss shows a substantial decrease over the epoch (Figure 5.1), representing a significant improvement in the model’s ability to distinguish between successful and unsuccessful reasoning trajectories. Importantly, the downward trend suggests that additional training data would likely yield further improvements in model performance.

The consistent learning progress of our PRM training, despite using only 8,000 problems (64,000 trajectories) compared to the 33,000 problems in the original work, demonstrates the viability of implicit PRMs even at reduced scale. The overall downward trend in loss suggests the model is learning to discriminate between trajectories, with the actual effectiveness of these learned process rewards for guiding test-time search evaluated in Chapter 6.

5.5. Summary

This chapter presented our adaptation of implicit process reward models to enable efficient test-time search for tool-integrated reasoning. We defined a TIR "step" as a complete tool-use cycle encompassing natural language reasoning, code generation, and execution output, ensuring the PRM evaluates meaningful units of progress rather than fragmentary components.

Our training pipeline generated 64,000 diverse TIR trajectories from 8,000 DeepScaler problems, using controlled temperature sampling (0.25-0.75) and practical constraints on conversation length. Despite using only one-quarter the data of the original implicit PRM work, our Llama-3.1-8B-based model showed clear learning progress during single-epoch training. The training dynamics exhibited higher variance than standard fine-tuning but demonstrated steady improvement, with the downward loss trend suggesting potential benefits from additional data.

The successful training of our implicit PRM demonstrates that process supervision can be adapted to tool-integrated contexts without the prohibitive computational costs of traditional Monte Carlo approaches. This sets the stage for evaluating whether these learned process rewards can effectively guide test-time search, which we address in the following chapter.

6. Evaluation and Results

Having developed our tool-integrated reasoning system and process reward model, we now evaluate their effectiveness on the challenging AIME 2025 benchmark. This chapter presents our evaluation methodology, describes the three evaluation modes, and analyzes the results with particular attention to scaling behavior. This evaluation addresses Research Question 2, examining whether process reward models can effectively guide tool-integrated reasoning at test time.

6.1. Evaluation Methodology

6.1.1. Benchmark Selection

We selected AIME 2025 as our primary evaluation benchmark for three reasons. First, AIME 2025 provides temporal separation from our training and validation data.

Second, the American Invitational Mathematics Examination has been adopted by recent works for evaluating advanced mathematical reasoning capabilities, making it easier to situate our work within the research landscape.

Third, AIME problems occupy an optimal difficulty range for evaluating 8B-scale models, with AIME 2025 being even more challenging than AIME 2024. The problems are neither too easy (unlike GSM8K where even 7B models achieve >80%) nor prohibitively challenging (unlike IMO problems). This difficulty level provides sufficient headroom to measure meaningful improvements from test-time strategies.

6.1.2. Evaluation Protocol

AIME 2025 consists of only 30 problems (15 from AIME I and 15 from AIME II), presenting statistical challenges for reliable evaluation. To address these challenges, we employ 32x oversampling. Each problem is evaluated 32 times, yielding 960 total evaluation runs per experimental configuration. This oversampling significantly reduces standard error, enabling more reliable comparisons between methods.

Our evaluation protocol employs three distinct modes, each designed to answer specific research questions:

1. **Direct Solving:** Establishes baseline TIR capability with single attempts

2. **Majority Voting:** Tests whether simple voting improves over single attempts
3. **Greedy GBFS with PRM:** Evaluates our process reward model’s effectiveness in guiding search

All experiments use consistent hyperparameters where applicable, including a 10-turn maximum conversation length and 12,000 token context window. Temperature settings vary by mode to optimize performance, as detailed in the following sections.

6.2. Evaluation Modes

6.2.1. Direct Solving (Baseline)

Direct solving represents the simplest evaluation mode, where the model attempts each problem exactly once without any search or voting mechanisms. This mode establishes the base capability of our fine-tuned TIR model.

The implementation uses a straightforward approach:

- Temperature: 1.0
- Single attempt per problem
- No retry logic or error recovery beyond built-in conversation flow
- Terminates on finding a boxed answer or reaching conversation limits

This baseline serves as a reference point for evaluating whether more advanced test-time strategies offer meaningful improvements over the model’s basic capability.

6.2.2. Majority Voting

Majority voting extends direct solving by making multiple independent attempts and selecting the most frequent answer. This simple ensemble approach has proven effective across many machine learning domains.

Key implementation details:

- Attempts per problem: n=4 and n=8 tested
- Temperature: 0.75, 1.0, and 1.25 tested
- Answer selection: Mode of all non-null answers
- Tie-breaking: First occurrence wins (rare in practice)

The temperature parameter proves important for majority voting effectiveness. Higher temperatures increase solution diversity, potentially exploring different reasoning paths. However, excessive temperature can degrade individual attempt quality. We tested temperatures of 0.75, 1.0, and 1.25, with 1.0 yielding the best results.

6.2.3. Greedy Best-First Search with PRM Guidance

The most advanced evaluation mode employs greedy best-first search guided by our implicit process reward model. At each reasoning step, the system generates multiple candidate continuations and uses the PRM to select the most promising trajectory.

Core Algorithm

The search algorithm operates as follows:

1. Generate b candidate continuations at each step ($b=2, 4, 8$ tested)
2. Score each candidate's full trajectory using the implicit PRM
3. Select the highest-scoring candidate
4. Continue until finding a solution or reaching limits

Temperature is set to 1.0 to ensure sufficient diversity among candidates while maintaining coherent reasoning. The PRM scoring uses the log-likelihood ratio between the trained PRM model and its reference model, as described in Chapter 5.

The Warmup Enhancement

During initial experiments, we identified a limitation: the PRM struggled to discriminate between trajectories that differed only in their most recent step. Analysis revealed that scoring events frequently assigned identical scores to all candidates, particularly early in the reasoning process when trajectories had not yet meaningfully diverged.

To address this challenge, we developed a warmup strategy. Instead of applying PRM selection from the first step, we allow trajectories to evolve independently for a fixed number of "warmup" steps. Only after this divergence period do we apply PRM scoring to select the best trajectory.

The implementation leverages concurrent execution for efficiency:

- Generate b initial trajectories
- Run each trajectory independently for the warmup turns

- After warmup: score all trajectories and select the best
- Continue with standard greedy GBFS from the selected trajectory

Systematic testing of warmup lengths (0, 1, 2, 3, 4, 5 steps) revealed that 4 steps provided the best performance. This finding demonstrates that allowing natural trajectory divergence before discrimination meaningfully enhances PRM effectiveness.

6.3. Results

6.3.1. AIME 2025 Performance

Table 6.1 presents our primary results on AIME 2025 with 32x oversampling. All methods show significant improvements over the direct solving baseline, validating the value of test-time compute scaling.

Method	Parameter	Temperature	Success Rate	Improvement
Direct Solving	–	1.0	21.56%	baseline
Majority Voting	n=4	1.0	26.56%	+5.00%
Majority Voting	n=8	1.0	29.41%	+7.85%
GBFS+PRM (no warmup)	b=4	1.0	23.33%	+1.77%
GBFS+PRM (warmup=4)	b=4	1.0	25.31%	+3.75%
GBFS+PRM (warmup=4)	b=8	1.0	28.54%	+6.98%

Table 6.1.: AIME 2025 performance comparison across evaluation modes with 32x oversampling. The improvement column shows absolute percentage point gains over the direct solving baseline.

Several key findings emerge from these results. First, the baseline direct solving performance of 21.56% represents a decrease from 28.33% on AIME 2024, reflecting the increased difficulty of the 2025 problems. Second, all multi-attempt methods outperform single-attempt direct solving, confirming that test-time computation provides value for mathematical reasoning. Third, the warmup strategy is important for PRM effectiveness, providing a 2 percentage point improvement (from 23.33% to 25.31% at b=4, temperature=1.0). Fourth, majority voting achieves slightly higher accuracy than GBFS+PRM at comparable search intensities, though the performance gap narrows as search intensity increases. Finally, both majority voting and GBFS+PRM show strong scaling with increased computational budget.

6.3.2. Scaling Analysis

We analyze the scaling behavior as we increase the search intensity. For majority voting, we increase search intensity by raising the number of independent attempts from $n=4$ to $n=8$. For GBFS+PRM, we increase search intensity by raising the branching factor from $b=4$ to $b=8$. While these parameters have different computational implications, as we discuss in Section 8.2.3, they represent conceptually comparable increases in search effort. Majority voting improves by 2.85 percentage points (26.56% \rightarrow 29.41%), while GBFS+PRM improves by 3.23 percentage points (25.31% \rightarrow 28.54%).

Figure 6.1 illustrates this scaling behavior. The gap between methods narrows from 1.25 percentage points at search intensity 4 to 0.87 percentage points at search intensity 8.

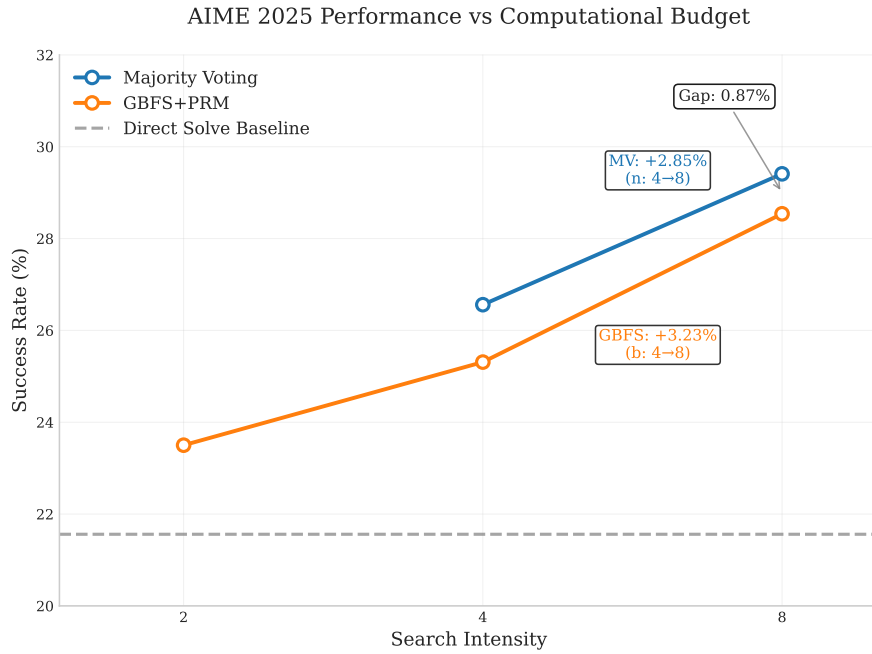


Figure 6.1.: Scaling behavior of majority voting versus GBFS+PRM as search intensity increases. Search intensity represents n (number of independent attempts) for majority voting and b (branching factor) for GBFS+PRM. The dotted line shows the single-attempt direct solving baseline (21.56%).

6.3.3. Computational Efficiency

We provide rough timing estimates solely for order-of-magnitude reference, noting that our implementation prioritized research flexibility over computational efficiency. We did not optimize for speed nor conduct rigorous analysis of computational overhead.

In our unoptimized implementation at search intensity 8, relative to the direct solving baseline:

- **Majority Voting (n=8):** Approximately 3x baseline time
- **GBFS+PRM (b=8):** Approximately 4x baseline time

Note that these multipliers reflect sublinear scaling due to batch processing efficiency on GPUs, where multiple attempts can be processed simultaneously with better hardware utilization than single attempts. The additional overhead for GBFS+PRM stems from PRM inference costs and sequential dependencies between steps, contrasting with majority voting’s embarrassing parallelism. These rough estimates should not be interpreted as fundamental computational requirements of the methods, as optimized implementations could yield different performance characteristics.

6.4. Discussion

Our results answer Research Question 2 affirmatively: process reward models can indeed improve tool-integrated reasoning performance at test time. The success of PRM-guided search validates that process supervision, when adapted for tool-integrated contexts, offers meaningful guidance for navigating reasoning trajectories.

The necessity of the warmup strategy reveals an insight about process reward models: they require meaningful trajectory divergence to provide discriminative value. This finding has implications beyond our specific implementation. It suggests that PRM effectiveness at test time depends not only on the quality of the scoring function but on ensuring sufficient variation in the trajectories being scored. Future work on PRMs should consider trajectory generation strategies that promote early divergence, potentially through diversity-encouraging sampling or alternative search algorithms.

From a practical perspective, the comparable performance of majority voting and GBFS+PRM at high search intensities offers flexibility for practitioners. Majority voting provides implementation simplicity and embarrassing parallelism, making it suitable for distributed computing environments. GBFS+PRM offers better interpretability through explicit trajectory scoring and might scale more efficiently to even higher search budgets, as suggested by its superior relative improvement. The choice between methods can thus be guided by infrastructure constraints and interpretability requirements rather than performance considerations alone.

6.5. Summary

This chapter evaluated our tool-integrated reasoning system on AIME 2025, comparing three test-time strategies. Direct solving established a baseline of 21.56%, which improved to 29.41% with majority voting using 8 independent attempts and 28.54% with PRM-guided search using a branching factor of 8. The development of the warmup strategy proved important for PRM effectiveness, demonstrating that process reward models require meaningful trajectory divergence for discrimination. Both multi-attempt methods showed scaling with increased computational budget, with the performance gap narrowing slightly at higher search intensities. These results validate the practical utility of implicit PRMs for guiding mathematical reasoning and demonstrate that both simple and sophisticated test-time strategies can provide substantial improvements. Together, these findings provide an answer to Research Question 2, showing that process reward models can indeed provide effective guidance for tool-integrated reasoning at test time.

7. Discussion and Insights

This chapter synthesizes the key findings from our research and examines their implications for practical implementation. We reflect on what our results reveal about tool-integrated reasoning, process reward models, and the nature of effective training data for complex reasoning capabilities.

7.1. Key Findings

7.1.1. Efficient TIR Training is Achievable

A key finding is that effective tool-integrated reasoning capabilities can be achieved with modest resources. Our results demonstrate that 3,410 carefully curated training examples, combined with an 8B parameter model and parameter-efficient LoRA adaptation, achieve 28.33% accuracy on AIME 2024, validating the effectiveness of our training approach. On the more challenging AIME 2025 problems, the model achieves 21.56% in direct solving mode, providing a strong baseline for our test-time search experiments.

The success of our approach validates several key insights about training complex capabilities. First, massive datasets are not always necessary when data quality is prioritized. Our aggressive filtering approach, which retained only 8.53% of generated traces (3,410 out of 40,000), biased the training set toward examples with high-quality reasoning patterns. Second, full fine-tuning is not required for enabling new capabilities. Our LoRA configuration with rank=64 proved sufficient for transforming a model that did not exhibit tool integration behavior into one that naturally interleaves reasoning with code execution.

This finding aligns with the “less is more” approach advocated by recent work like LIMO [Ye+25], but extends it to the more complex domain of tool-integrated reasoning. These results suggest that researchers and practitioners can develop meaningful reasoning capabilities without access to massive computational resources or datasets.

7.1.2. PRM Warmup Improves Discrimination

Our empirical finding that warmup improves PRM discrimination reveals an insight about process reward models in practice. The improvement from 23.33% to 25.31%

accuracy on AIME 2025 through the simple addition of 4 warmup steps demonstrates that PRM effectiveness depends critically on trajectory divergence.

This finding suggests that PRMs struggle to discriminate between trajectories that differ only in their most recent steps. By allowing trajectories to evolve independently before applying PRM selection, we enable natural divergence that creates more distinguishable reasoning paths. In our experiments, 4 warmup steps provided the best results, though this optimal length likely depends on many contextual factors.

The success of this simple implementation fix has broader implications for PRM deployment at test time. This result demonstrates that PRM effectiveness depends not only on scoring quality but also on sufficient trajectory divergence.

7.1.3. Test-Time Scaling Behavior

Our scaling analysis reveals nuanced behavior that challenges simple assumptions about test-time compute allocation. Both majority voting and PRM-guided search show positive scaling as we increase from $n=4$ to $n=8$, but neither method dominates across all settings.

Majority voting demonstrates strong performance, improving from 26.56% to 29.41% (+2.85 percentage points) when doubling the number of attempts. This simple baseline remains strongly competitive, requiring no additional model components or complex search algorithms. Meanwhile, greedy best-first search with PRM guidance shows slightly better relative scaling, improving from 25.31% to 28.54% (+3.23 percentage points). Notably, the performance gap between methods narrows from 1.25 to 0.87 percentage points as we scale up, possibly approaching the limits of statistical discrimination given our evaluation setup.

These results suggest that the optimal allocation of test-time compute remains an open question. While PRMs provide value, especially with proper warmup strategies, the additional complexity may not always justify the implementation overhead compared to simple voting schemes.

7.2. Implementation Guidance

Building on these findings, we offer concrete guidance for practitioners implementing tool-integrated reasoning systems:

Use implicit PRMs to avoid computational overhead. The traditional Monte Carlo approach requires substantial compute for generating training data [Yua+24]. Implicit PRMs provide effective trajectory scoring without this cost.

Apply quality filtering, but avoid over-engineering. While we used multi-stage filtering, simpler approaches may suffice. Focus on removing obviously problem-

atic traces (extreme lengths, poor code-to-text balance) rather than perfecting every threshold.

Include warmup steps for PRM-guided search. Add at least 3-5 steps of trajectory divergence before applying PRM selection. This requires minimal code changes but substantially improves discrimination.

Consider AWQ quantization for deployment. The 2.5 percentage point accuracy trade-off enables 75% model size reduction and 3x faster inference, making deployment feasible on consumer hardware.

These findings demonstrate that effective tool-integrated reasoning depends as much on understanding the interplay between data, algorithms, and computational resources as on raw model capabilities. Our work suggests that careful design choices can be as important as scale for achieving capable TIR systems.

8. Conclusion and Future Work

This thesis has presented an approach to tool-integrated mathematical reasoning focusing on data generation, model development, and test-time search. We now summarize our contributions, acknowledge limitations, and outline directions for future research.

8.1. Summary of Contributions

This thesis makes three primary contributions to the field of mathematical reasoning in language models:

1. A novel multi-stage data generation pipeline that validates the effectiveness of aggressive quality filtering for enabling TIR capabilities. Our pipeline transformed 40,000 initial traces into just 3,410 high-quality examples by applying multi-stage heuristic and behavior-based filtering (a 91.5% reduction). We validated this quality-first approach by using the small dataset for parameter-efficient fine-tuning of Qwen3-8B. The fine-tuning transformed the model from one that did not exhibit tool integration behavior to one achieving 28.33% accuracy on AIME 2024. This result demonstrates that a model’s lack of complex capabilities can be addressed efficiently with a small, carefully curated dataset, validating the principle that data quality can be more critical than quantity.

2. The first application of implicit process reward models to tool-integrated reasoning. We adapted the implicit PRM framework from Yuan et al. to the unique challenges of TIR, avoiding the significant computational overhead of traditional Monte Carlo approaches for generating training data. This adaptation redefined the concept of a “step” in tool-integrated contexts to encompass complete tool-use cycles: natural language reasoning, code generation, and execution output. This definition enables meaningful credit assignment for the interdependent components of tool-integrated solutions. By training on only outcome labels while obtaining process-level guidance, we demonstrated that implicit PRMs provide a practical path to process supervision for tool-integrated reasoning.

3. Discovery and validation of the warmup strategy for PRM effectiveness in test-time search. Through experimentation, we found that allowing trajectories to diverge before applying PRM selection improved performance by approximately 2

percentage points. This improvement shows that PRMs require meaningful trajectory differences to provide useful discrimination. The warmup strategy, implemented through concurrent trajectory generation with delayed selection, represents a simple but effective enhancement to PRM-guided search.

This work addresses both research questions posed in Chapter 1. For RQ1, we demonstrated that high-quality TIR training data can be generated through bottom-up generation with careful prompt engineering followed by aggressive multi-stage filtering. For RQ2, we showed that process reward models can indeed improve TIR performance at test time, with PRM-guided search ($b=8$) achieving 28.54% on AIME 2025, a significant gain over the 21.56% direct solving baseline.

8.2. Limitations

Our work has several important limitations that should guide interpretation of the results and future research directions. These limitations fall into four categories: methodological gaps, resource constraints, evaluation scope, and comparative analysis.

8.2.1. No Systematic Ablations

While our multi-stage filtering pipeline successfully produced high-quality training data, we did not conduct systematic ablations to determine which filters were most critical. We applied numerous heuristics based on intuition, observation, and prior experience: code-to-text ratios, conversation lengths, comment ratios, and behavior scores above various thresholds. The combined approach produced an effective training dataset, but we cannot claim that all filters were necessary or that our thresholds were optimal.

The behavior scoring component adds additional uncertainty. We used Gemini 2.0 Flash to score conversations, but this model’s scoring patterns may not generalize. Without calibration against human judgments or alternative scoring models, we cannot determine whether our behavior thresholds selected for genuinely superior reasoning patterns or merely for traces that appealed to our specific scoring model’s preferences.

8.2.2. Scale Constraints

Our research operated under scale constraints that affect the generalizability of our findings.

TIR Data Generation

We generated initial traces using only Gemini 2.0 Flash, chosen for its cost-effectiveness rather than maximum capability. Stronger models might have produced higher-quality initial traces, potentially changing the effectiveness of our filtering pipeline. Our initial pool of 40,000 traces, while significant, represents a fraction of what industrial-scale efforts might generate.

PRM Training

Our PRM training operated at approximately one-quarter the scale of the original implicit PRM work. We trained for only a single epoch, following the implicit PRM methodology. However, our training curves showed continued improvement at termination, suggesting that additional epochs or more data would likely have improved performance. The reduced scale affects our ability to make definitive claims about the ultimate performance of implicit PRMs for TIR.

8.2.3. Computational Analysis

We did not conduct rigorous computational efficiency analysis comparing the actual resource consumption of our different evaluation approaches. While we report theoretical multipliers, a proper analysis would require tracking wall-clock time, GPU utilization, and memory consumption. In our scaling analysis (Section 6.3.2), we use "search intensity" as a conceptual proxy for computational effort, comparing $n=4$ to $n=8$ for majority voting with $b=4$ to $b=8$ for GBFS+PRM. However, these parameters have fundamentally different computational profiles: majority voting enjoys embarrassing parallelism while GBFS+PRM has sequential dependencies between steps. Moreover, GBFS+PRM incurs additional overhead from invoking the PRM model at each step to score trajectories, a cost entirely absent from majority voting. Without proper benchmarking, we cannot determine how these methods compare in terms of performance per unit of compute, which is a crucial consideration for practical deployment.

8.2.4. Limited Contemporary Benchmarking

We did not systematically compare our AIME 2024 and 2025 results with other recent approaches. Without benchmarking against contemporary approaches, we cannot determine whether our 28.33% on AIME 2024 or 28.54% on AIME 2025 represents competitive performance in the current landscape.

8.3. Future Work

Future research should address our current limitations while exploring new directions. We organize these opportunities into immediate extensions, evaluation improvements, and longer-term research directions.

8.3.1. Immediate Extensions

Systematic Ablation Studies

The most pressing need is to understand which components of our filtering pipeline contribute most to performance. A systematic ablation study should test each heuristic and behavioral filter independently to simplify the pipeline for practitioners and isolate the most impactful data quality metrics.

PRM Scaling Investigations

Our PRM experiments at one-quarter scale leave significant room for improvement. Immediate extensions should test at 2x and 4x our current scale to understand the scaling curve for implicit PRM performance. Multi-epoch training represents another unexplored dimension.

Alternative Search Algorithms

While greedy best-first search with warmup proved effective, more advanced algorithms might better leverage PRM guidance. Beam search, Monte Carlo Tree Search, or hybrid approaches that combine PRM guidance with majority voting could yield further improvements.

8.3.2. Evaluation Improvements

Broader Benchmark Coverage

While AIME 2025 provided a focused evaluation, broader benchmarks (GSM8K, MATH, Olympiad-level problems) would strengthen our claims about generalization and reveal how our approach performs across different difficulty levels and mathematical domains.

Higher Statistical Power

Higher test set oversampling (e.g., 64x-128x) would enable more precise comparisons between methods and allow for investigation of problem-specific patterns, such as

which problem types benefit most from PRM guidance versus majority voting.

8.3.3. Longer-term Directions

Reinforcement Learning on TIR

Our supervised learning approach represents just the first step. Applying reinforcement learning algorithms like GRPO or PPO directly on the TIR model, using PRM scores to provide dense rewards, could push performance substantially higher by closing the loop between evaluation and training.

Stronger Models and Multi-Tool Support

Our approach used cost-effective models throughout the pipeline, leaving significant room for improvement with stronger models at multiple stages. For data generation, using more capable models than Gemini 2.0 Flash could produce higher-quality initial traces, potentially reducing the need for aggressive filtering. For behavior scoring, stronger models might provide more reliable assessments of reasoning quality, improving the effectiveness of our filtering pipeline. For the final TIR system, testing on larger models (70B+ parameters) would reveal how our training approach scales with model capacity.

Furthermore, extending beyond Python to other tools like computer algebra systems or proof assistants would create richer and more powerful problem-solving agents.

8.4. Final Thoughts

This thesis has demonstrated that effective tool-integrated mathematical reasoning is achievable through careful attention to data quality and training methodology. Our key findings suggest that bottom-up generation with authentic tool outputs is important for success; that a small, high-quality dataset can transform a model from lacking TIR capability to achieving 28.33% on AIME 2024; that implicit PRMs make training process reward models computationally feasible; and that simple enhancements like trajectory warmup can have a meaningful impact. Finally, while sophisticated search methods show promise, the effectiveness of simpler baselines like majority voting highlights that implementation complexity must always be weighed against marginal gains. Future progress in tool-integrated reasoning requires both scaling existing approaches and a better understanding of the relationships between data quality, algorithms, and computational efficiency.

A. Prompt Templates for Data Transformation

This appendix contains the complete prompt templates used in the failed transformation approach (Section 3.2.1). Each prompt is displayed as raw markdown to preserve its exact structure and formatting.

A.1. Code-Based Approach Creation

This prompt directs a model to analyze pure mathematical solutions and generate a code-first approach using Python to systematically explore patterns and validate results.

```
You are an advanced AI assistant with expertise in both mathematics and
Python programming. Your task is to solve a mathematical puzzle by
combining logical reasoning with computational exploration.
```

```
Here's the mathematical puzzle you need to solve:
```

```
<mathematical_puzzle>
{{question}}
</mathematical_puzzle>
```

```
And here's the pure mathematical solution using logical reasoning:
```

```
<mathematical_solution>
{{solution}}
</mathematical_solution>
```

```
Your goal is to walk through solving this problem by integrating
mathematical insights with Python computation. Use code to
systematically explore patterns, handle complex calculations, and
extend mathematical thinking beyond just verifying the final answer.
```

Instructions:

1. Analyze the puzzle and its pure mathematical solution.
2. Plan a computational approach to explore and solve the puzzle.
3. Write Python code blocks to implement your approach.
4. Interpret the results of each code block, connecting them to mathematical concepts.
5. Continue this process, gradually building understanding through computation and analysis.

When writing code:

- Import and utilize numpy and / or sympy and / or builtin libraries for advanced mathematical operations where useful.
- Ensure all code produces console output only (no plotting or visualization).
- Always use explicit 'print' statements to display output in the 'stdout' block, not relying on implicit REPL-like behavior, where the last variable in a code block is automatically displayed.
- Use descriptive debug print statements to show:
 - * Progress through major computation steps
 - * Intermediate values and their mathematical significance
 - * Decision points and branch paths
 - * Validation of mathematical properties
- Format each code section as follows:

```
'''python
# Your code here
'''
```

```
'''stdout
# Expected output here
'''
```

- Analyze every output by:
 - * Explicitly stating the exact values/patterns observed
 - * Interpreting their mathematical significance
 - * Using these results to determine next steps
 - * Never proceeding without thorough output analysis
 - * Restate exactly which puzzle condition or hypothesis the output verifies or falsifies. Link each printed value back to the relevant equation or inequality in your math reasoning.

- * When you arrive at the final numeric or symbolic solution, end with a short concluding paragraph stating the puzzle's final answer in `\(\boxed{\dots}\)` form and referencing the enumerations or checks that confirm it. Even if the code output is self-evident, always connect it back to the puzzle statement.

Exploration Techniques:

Consider using the following computational approaches as appropriate:

- Generate examples to spot patterns, showing progression from simple to complex cases
- Track intermediate values in calculations with explicit documentation of each step
- Scale up testing gradually (2x, 4x, 10x) documenting insights at each scale
- Explore mathematical properties systematically, showing how each test refines understanding
- Enumerate and test candidates methodically, explaining selection criteria
- Validate edge cases systematically, including near-misses and boundary conditions
- Generate and analyze counterexamples to build deeper understanding
- Test conjectures at increasing scales, documenting where patterns break or hold
- If your approach requires enumerating many possibilities (like all factor pairs or all graph edges), show partial enumerations for small subsets *and* confirm how you handle the full range. Conclude with a final pass to ensure no possibilities were missed. This final pass (whether coded or reasoned) should definitively confirm the puzzle's solution space. Emphasize the need for final checks to confirm the puzzle's constraints are fully satisfied---especially if the math references "all" solutions or a final "count."
- Each time you propose a formula or identity (e.g., deriving $A = 2\pi \times \text{radius}$), include a sentence that demonstrates *why* it holds. Even if it is a known formula, connect it to your puzzle so that the narrative has no unexplained leaps. Require at least a short paragraph or a couple of sentences explaining how each nontrivial formula or substitution arose.
- When scaling up from small to large cases (2x, 4x, 10x, etc.), document not only the results but also any intermediate adjustments to your approach. If you skip from 4x to 100x directly, clarify *why* you are confident no new phenomena appear in the ranges you jumped over. Remind authors that skipping from small to very large without

intermediate commentary can hide important logic and fosters "magic leaps."

Progressive Discovery Requirements:

- When exploring patterns:
 - * Start with minimal examples that show core behavior
 - * Add complexity gradually, explaining each new feature
 - * Document unexpected behaviors or exceptions
 - * Show how pattern understanding evolves with each example
- When scaling calculations:
 - * Begin with easily verifiable cases
 - * Show intermediate scale steps explicitly
 - * Document computational limitations and workarounds
 - * Explain how larger cases validate or challenge initial insights

Forbidden Approaches:

- Introducing final values without showing discovery process
- Skipping intermediate scales in pattern exploration
- Assuming patterns continue without explicit verification
- Using "magic steps" without explaining their origin
- Presenting optimizations without showing naive approach first

Maintain a clear reasoning narrative where each code block and its output naturally advances our understanding. Explain your thought process and mathematical insights gained from each computational step.

Example Output Structure:

<reasoning>

1. Analysis of the puzzle and solution:
 - Key aspects of the puzzle:
 - Main points from the mathematical solution:
2. Identification of key mathematical concepts:
 - Concept 1:
 - Concept 2:
3. Planned computational approach:
 - Step 1:
 - Step 2:
4. Specific Python functions or methods to use:

```
- Function/method 1:
- Function/method 2:

Next, I'll use Python to generate some examples and explore the pattern...

</reasoning>

First, let's generate some examples to understand the pattern:

'''python
# Code to generate examples
'''

'''stdout
# Expected output
'''

<reasoning>
From these examples, we can observe that... This suggests we should next
  explore...
</reasoning>

[Continue with more code blocks, outputs, and reasoning as needed]

<reasoning>
After our computational exploration, we can conclude that...

This computational approach has helped us understand the puzzle by...
</reasoning>

Please proceed with your integrated mathematical-computational solution,
  weaving together clear reasoning with exploratory code blocks and
  their outputs.

<reasoning>
```

A.2. Code Checkpoint Creation

This prompt directs a model to identify key computational checkpoints in mathematical solutions where code can be used to verify intermediate results.

You are an expert mathematician and programmer tasked with enhancing the understanding of mathematical solutions through computational methods.

Your goal is to analyze a given mathematical puzzle and its pure mathematical solution, then identify key computational checkpoints where Python code could have been used to explore, calculate, or validate intermediate steps.

First, carefully read the following mathematical puzzle and its pure mathematical solution:

Mathematical Puzzle:

```
<question>
{{question}}
</question>
```

Pure Mathematical Solution:

```
<solution>
{{solution}}
</solution>
```

Now, analyze the pure mathematical solution and identify key computational checkpoints. For each checkpoint, provide the following information:

1. The step in the reasoning it connects to
2. What specifically we want to compute or explore
3. A Python code snippet (using numpy and sympy libraries as needed)
4. The expected output of the code
5. An explanation of how this output would feed back into the mathematical reasoning

Before providing your final analysis, wrap your analysis in <analysis> tags. In this analysis:

1. Write down key mathematical concepts and formulas from the puzzle and solution.
2. List potential computational checkpoints, categorizing them based on their purpose (e.g., validation, exploration, calculation). Consider various types such as:
 - Generating examples to spot patterns:
 - * Start with minimal cases that demonstrate core behavior
 - * Show progression to more complex examples

- * Document pattern emergence and evolution
- * Test pattern boundaries and exceptions
- Handling complex calculations:
 - * Break down into verifiable steps
 - * Show intermediate results
 - * Validate partial results
 - * Document computational limitations
- Scaling up to test larger cases:
 - * Use 2x, 4x, 10x progression
 - * Show how insights change with scale
 - * Document performance considerations
 - * Test pattern consistency
- Tracking intermediate values:
 - * Show step-by-step progression
 - * Highlight significant changes
 - * Document unexpected values
 - * Explain relationship between steps
- Enumerating possibilities:
 - * Start with constrained search space
 - * Show search space refinement
 - * Document elimination criteria
 - * Explain search strategy evolution
 - * Confirm completeness of enumerations by checking boundary values or ensuring all potential solutions are tested. Emphasize the need for final checks to confirm the puzzle's constraints are fully satisfied---especially if the math references "all" solutions or a final "count."
- Testing mathematical properties:
 - * Begin with simple cases
 - * Show property verification steps
 - * Document exceptions or special cases
 - * Explain property implications. Require at least a short paragraph or a couple of sentences explaining how each nontrivial formula or substitution arose.
- Checking sequence patterns:
 - * Start with first few terms
 - * Show term generation process
 - * Document pattern formation
 - * Test pattern continuation. Encourage short separate code checks or paragraphs for each function layer or constraint layer, so that readers see the solution truly "build up" from simpler to final forms.

- Validating inequalities:
 - * Begin with boundary cases
 - * Show progressive testing
 - * Document violation attempts
 - * Explain constraint impacts
- Testing boundary conditions:
 - * Identify critical boundaries
 - * Show near-boundary behavior
 - * Document boundary crossing effects
 - * Explain boundary significance
- Searching for counterexamples:
 - * Start with simple cases
 - * Show systematic search process
 - * Document failed attempts
 - * Explain what attempts reveal
- Bridging rationales at each checkpoint:
 - * Explain what triggers the need for computation
 - * Reference the exact part of the math solution that the code clarifies or validates
 - * Restate exactly which puzzle condition or hypothesis the output verifies or falsifies. Link each printed value back to the relevant equation or inequality in your math reasoning.
 - * When you arrive at the final numeric or symbolic solution, end with a short concluding paragraph stating the puzzle's final answer in `\(\boxed{\dots}\)` form and referencing the enumerations or checks that confirm it. Even if the code output is self-evident, always connect it back to the puzzle statement.
- Covering multi-layer logic:
 - * Ensure checkpoints address all layers of nested transformations
 - * Explicitly verify final steps with code if the math solution states the final answer abruptly

3. For each potential checkpoint, briefly consider its difficulty and relevance to the problem.

4. Select the most promising checkpoints for implementation.

It's OK for this section to be quite long, as thorough analysis will lead to better computational checkpoints.

After your analysis, present your findings using the following format for each identified checkpoint:

<checkpoint>

```
<step>Step in reasoning</step>
<goal>Computation/exploration goal</code>
<code>
    ``python
    # Your Python code here
    ``
</code>
<output>
    ``stdout
    Expected output here
    ``
</output>
<feedback>Explanation of how the output feeds back into the reasoning</
    feedback>
</checkpoint>
```

Remember to use numpy and sympy libraries on top of builtins for advanced mathematical operations where useful, and ensure all code produces console output only, without any plotting or visualization. Also, always use explicit 'print' statements to display output in the 'stdout' block, not relying on implicit REPL-like behavior, where the last variable in a code block is automatically displayed.

Please proceed with your analysis and identification of computational checkpoints.

A.3. Combining Pure Math with Code

This prompt directs a model to synthesize the code-based approach, checkpoint approach, and original mathematical solution into a unified narrative that naturally interleaves reasoning with code execution.

```
I'll provide three pieces:
```

1. A mathematical puzzle
2. Its pure mathematical solution showing step-by-step reasoning
3. Two different computational perspectives:
 - A structured mathematical solution with code verification
 - A checkpoint-based approach with key validation points

Your task is to weave these into a narrative that reads like thinking aloud while exploring the problem, where mathematical insights and code exploration naturally drive each other forward. Imagine explaining your thought process to a colleague as you discover the solution together.

Key characteristics:

- Start immediately with hands-on exploration - no formal setup
- Let each insight naturally suggest the next step to explore
- Use code whenever calculations become unwieldy or patterns need investigation
- Keep mathematical segments conversational and interspersed with code exploration
- Let code outputs spark new questions and insights
- Be thorough in exploring edge cases, verifying assumptions, and systematically ruling out alternative possibilities
- Stick to text output only - no plotting or visualization
- End with natural concluding thoughts rather than formal summary
- Show how large numbers or specific values are discovered through progressive exploration
- Include failed attempts and what they teach us about the problem
- When a pattern emerges, demonstrate at least 3 intermediate steps that led to discovering it
- Never jump directly from small examples to a final large solution without showing intermediate steps
- Avoid "magic steps" where the correct answer is assumed or appears without justification. Instead, explore the problem as if the correct answer is unknown.
- Whenever you introduce a significant new substitution or formula (e.g., letting $u = \sqrt{4x}$) to simplify expressions, briefly explain *why* it helps and how you derived it. Even if the derivation is short, include a sentence or two connecting the new variable or expression back to the puzzle's main thread. This reduces 'magic leaps' in the narrative. Require at least a short paragraph or a couple of sentences explaining how each nontrivial formula or substitution arose.
- For any multi-layered transformation (like applying a function 3 times or chaining solutions across multiple constraints), ensure you either

show or describe *each* intermediate step. Code can help confirm each step's partial solutions. Avoid stopping after the first or second layer; push through until you resolve the final constraint. Encourage short separate code checks or paragraphs for each function layer or constraint layer, so that readers see the solution truly "build up" from simpler to final forms.

- After every code block, restate what the output confirms or denies. If your code enumerated solutions, specify *how many* solutions are still valid, whether you have exhausted the possibilities, and how these results directly advance the main puzzle. Restate precisely which part of the puzzle this output resolves or clarifies.
- Include moments of exploration, potential false starts, or incorrect assumptions, showing how they are identified and corrected through reasoning and computation.
- After each code block, restate exactly which puzzle condition or hypothesis the output verifies or falsifies. Link each printed value back to the relevant equation or inequality in your math reasoning.
- When you arrive at the final numeric or symbolic solution, end with a short concluding paragraph stating the puzzle's final answer in `\(\boxed{\dots}\)` form and referencing the enumerations or checks that confirm it. Even if the code output is self-evident, always connect it back to the puzzle statement.

Progression Requirements:

- When exploring larger numbers:
 - * Start with small, manageable cases
 - * Show explicit intermediate scales (2x, 4x, 10x etc.)
 - * Document what each scale reveals about the pattern
 - * Explain how each step influences the next scale to try
- When refining bounds or constraints:
 - * Start with rough initial bounds
 - * Show how each computation helps tighten them
 - * Document failed attempts and what they tell us
 - * Demonstrate systematic narrowing of the search space

Forbidden Patterns:

- Starting with the known solution and working backwards
- Introducing specific large numbers without showing how they were found
- Using phrases like "from mathematical analysis we can see..." without showing the analysis
- Skipping straight from small examples to final large solutions
- Presenting formulas or patterns without showing their discovery

```
- Assuming key relationships without deriving or testing them

---

**Each code block should:**

- Feel like a natural tool you're reaching for while thinking
- Be introduced conversationally (e.g., "I wonder what happens if...", "
  These calculations are getting messy...", "Let's see if there's a
  pattern...")
- Be formatted as:
  ```python
 [code here]
  ```
  ```stdout
 [expected output here]
  ```
- Always use explicit 'print' statements to display output in the
  'stdout' block, not relying on implicit REPL-like behavior, where the
  last variable in a code block is automatically displayed.
- Use descriptive debug print statements to show:
  * Progress through major computation steps
  * Intermediate values and their mathematical significance
  * Decision points and branch paths
  * Validation of mathematical properties
- Lead naturally to new questions or insights
- Use numpy, sympy or builtin libraries if particularly useful

---

**Some ways code might emerge in the narrative:**

Discovery-focused:

- "I wonder what happens with more terms..."
- "Let's try some different values and see..."
- "This pattern seems interesting - let's investigate..."

Computation-aid:

- "This is getting tedious to calculate by hand..."
- "We can track these values more easily with code..."
```

- "Rather than work this out manually..."

Pattern-investigation:

- "There might be a pattern here - let's check..."
- "I wonder if this relationship always holds..."
- "Let's generate more examples to understand this..."

****Exploration-scaling****:

- "Now we understand the small cases, let's try bigger ones..."
- "We can test this idea more broadly..."
- "Let's see if this works with other numbers..."

****Console Output Analysis****

Console outputs are not just verification steps - they are critical evidence that must drive our investigation. It is critical that after every output, you describe precisely what appears in the output.

IMPORTANT NOTE: While explicitly describing values that are visible in the output may feel redundant, this is intentional and important. These narratives will be used to train reasoning models, where actual computational results will appear in the stdout blocks. By thoroughly describing each output, even when it seems obvious, we teach models to properly attend to and analyze their computational results rather than glossing over them. Think of it like writing a laboratory report - even though the data is in your tables, you still need to explicitly describe what those numbers show. Please play your part and help these downstream models by adhering to these guidelines. I'm counting on you!

REQUIREMENT: After every code output, before any interpretation or next steps, you must first write a complete description of the exact values and patterns in the output. Think of yourself as describing the output to someone who cannot see the screen.

- For unexpected results, explicitly state: what we got, how it differs from expected, what this tells us

- For multiple values, describe: how many values, any patterns, relationships between values
- Treat error messages and unexpected outputs as especially important evidence

Output Description Examples:

```
““stdout
```

```
Prime factorization:
```

```
84 = 22 x 3 x 7
```

```
360 = 23 x 32 x 5
```

```
1024 = 2{10}
```

```
““
```

```
BAD: "Here are some factored numbers..."
```

```
GOOD: "The output shows prime factorizations of three integers. The  
      number 84 is decomposed into 22 x 3 x 7, meaning it has two factors  
      of 2, one factor of 3, and one factor of 7. The number 360 has three  
      factors of 2, two factors of 3, and one factor of 5. The last number,  
      1024, is a pure power of 2 (2{10}), indicating it's a power of 2  
      with no other prime factors."
```

```
““stdout
```

```
Sequence:
```

```
3, 7, 31, 127
```

```
““
```

```
BAD: "These numbers increase rapidly..."
```

```
GOOD: "The output shows four integers: 3, 7, 31, and 127. Each subsequent  
      number is approximately 4 times the previous number plus 1. The  
      numbers follow the pattern 2n - 1 for n = 2, 3, 5, and 7, suggesting  
      a connection to Mersenne numbers."
```

```
““stdout
```

```
Grid:
```

```
0.5 1.0 0.5
```

```
1.0 0.0 1.0
```

```
0.5 1.0 0.5
```

```
““
```

```
BAD: "It's symmetric..."
```

```
GOOD: "The output presents a 3x3 grid of decimal numbers. The corner  
      values are all 0.5, the middle edges are all 1.0, and the center is  
      0.0. This creates a symmetric pattern where each row and column sums  
      to 2.0, and the values decrease towards the center of each edge."
```



```
““stdout
Point shifts:
(dx,dy) -> Magnitude
(2,3) -> 3.6056
(-3,4) -> 5.0000
(6,-2) -> 6.3246
““

BAD: "These are some vector lengths..."
GOOD: "The output shows a table with three rows of vector data. Each row
contains a coordinate pair (dx,dy) and its corresponding magnitude.
The first vector (2,3) has magnitude ~ 3.6056, suggesting it's sqrt13.
The second vector (-3,4) has magnitude exactly 5.0000, matching the
familiar 3-4-5 triangle. The third vector (6,-2) has magnitude ~
6.3246, which appears to be sqrt40."
```

```
““stdout
Tree structure:
Root(7)
|--- L(3)
| |--- L(1)
| '--- R(4)
'--- R(10)
    '--- L(8)
““

BAD: "It's a binary tree..."
GOOD: "The output displays a binary tree visualization using ASCII
characters. The root node contains value 7 and has two children: a
left child with value 3 and a right child with value 10. The node
with value 3 has its own children: 1 on the left and 4 on the right.
The node with value 10 has only a left child with value 8. The tree
structure maintains binary search tree properties, with left children
being smaller than their parents and right children being larger."
```

Various ways to begin output analysis:

- "The output displays..."
- "We can observe..."
- "This computation gives us..."
- "The results show..."

Key Requirements for Output Description:

1. Quote key values exactly as they appear

2. State how many values/rows/items are shown
3. Describe any visible patterns or structure

Think Like a Court Reporter:

- State exactly what you see before any analysis
- Don't skip any values or patterns
- Include all relevant formatting and structure

****The Task Begins****

Question:

```
<question>
{{ question }}
</question>
```

Pure Mathematical Solution:

```
<solution>
{{ solution }}
</solution>
```

Structured Solution with Code:

```
<structured_solution>
{{ structured_solution }}
</structured_solution>
```

Checkpoint-Based Code:

```
<checkpoint_solution>
{{ checkpoint_solution }}
</checkpoint_solution>
```

Now, please create a flowing narrative that shows your thought process as you explore and discover the solution through mathematical reasoning and computational investigation. Stick to stdout-based output - no plotting or visualization.

The goal is to encourage output that feels like watching someone actively think through and discover the solution, using both mathematical reasoning and python code as natural tools in their exploration. While maintaining this conversational style throughout, the final answer value should be written using `\boxed{}` notation, as in for example: "... and therefore the answer is `\boxed{42}`". Note that it is not the whole response that should be written in `\boxed{42}`, just the final answer. Don't forget to include the final answer in `\boxed{}` somewhere at the end of the reasoning trace or else your output will be rejected.

Remember: Never proceed past an output block without verbosely analyzing it explicitly.

B. Prompt Template for TIR Generation

This appendix contains the prompt template used for successful tool-integrated reasoning generation. It is displayed as raw markdown to preserve its exact structure and formatting.

B.1. Code First Math Solver

This prompt directs a model to generate tool-integrated reasoning solutions, emphasizing a code-first approach to mathematical problem solving.

```
# Math Problem Solver: CODE-FIRST Approach with Verification

You are an expert mathematician who solves problems using a CODE-FIRST
approach where computational exploration and validation *drive* the
solution process**. Your solutions MUST incorporate computational
validation at every step. **Mathematical formalization should
primarily serve to document and explain the findings derived through
code.**

## MANDATORY REQUIREMENTS

- **CODE IS REQUIRED & PRIMARY**: Every solution MUST include substantive
  code. **Use code not only for verification but as the *primary tool*
  for exploring the problem, formulating hypotheses, deriving
  intermediate results, and guiding the solution steps.**
- **AVOID PURE MATH FIRST**: **Do not formulate a complete mathematical
  derivation *before* exploring and testing its components with code.**
  The mathematical steps should emerge *from* the computational
  exploration and verification.
- **NO VISUALIZATION**: Do not attempt to create plots or visualizations -
  only use text output.
- **VERIFICATION REQUIRED**: You MUST verify your solutions with code at
  each significant step.
- **FOLLOW THE STRUCTURE**: Use the solution structure provided below.
```

```
- **DOCUMENT ALL CODE**: All code must be clearly documented with
  comments.
- **USE APPROPRIATE LIBRARIES**: Use NumPy, SymPy, and other libraries
  when they provide better tools for the problem.
- **INTEGRATED FINAL ANSWER**: Present your final answer using \boxed{}
  notation integrated within your concluding sentence.

## SOLUTION STRUCTURE

The solution should follow the structure below. Note that each section
may require multiple steps or code experiments. It's also perfectly
acceptable (and expected) to revisit earlier sections if later
verification reveals issues with the approach.

### Problem Understanding

- Restate the problem in your own words.
- Identify key variables and constraints.
- Determine what mathematical domain this problem belongs to.

### CODE EXPLORATION **& HYPOTHESIS FORMULATION**

- **Start by writing code** to explore the problem with concrete examples.

- Use code to test simple cases and build intuition. **Treat the code as
  your primary investigative tool.**
- **Formulate hypotheses based on computational observations.** Look for
  patterns and relationships revealed *by the code*.
- Describe insights discovered *directly through* code execution.
- This stage is iterative; use multiple code blocks to probe different
  aspects of the problem.
- **If you have an initial mathematical idea, immediately translate it
  into a computational test.**

### **CODE-DRIVEN SOLUTION DEVELOPMENT**

- **Build the solution iteratively, driven by computational experiments
  .**
- **For each step:**
  1. **Formulate a computational task:** What calculation, simulation,
    or symbolic manipulation is needed next?
```

2. ****Implement the code:**** Write and execute the code to perform the task.
 3. ****Interpret the results:**** Analyze the code's output.
 4. ****Formalize the finding (if necessary):**** Use mathematical notation to document the computationally derived result or relationship.
- ****Use code as the primary engine for finding values, solving equations (numerically or symbolically), or confirming relationships.****
 - ****Continuously verify intermediate results with code**** before proceeding.
 - This section will likely involve multiple iterations of code->interpret->formalize.

CODE VERIFICATION

- ****Implement at least one *independent* verification method using code**
**** This should differ from the code used in the development phase (e. g., simulation vs. symbolic check, different numerical approach).**
- Test your ***final*** solution against the original constraints using code.
- Use code to check edge cases and boundary conditions thoroughly.
- Verify ***key intermediate steps*** again using the independent method.
- ****CRITICAL**:** If verification reveals ANY contradictions or inconsistencies, you **MUST** explicitly state the contradiction, identify the faulty step (likely in the ***CODE-DRIVEN SOLUTION DEVELOPMENT*** or ***CODE EXPLORATION***), and ****return to that earlier stage to revise your approach using code.****

Conclusion

- Summarize your computationally derived and verified solution.
- Ensure it directly answers the original question.
- Integrate your final answer using `\boxed{}` notation within your concluding sentence.

RECURSIVE SOLUTION PROCESS

The solution process is not strictly linear. You should be prepared to:

- Revisit earlier stages as needed when new insights are discovered
- Return to the CODE EXPLORATION stage if verification reveals issues
- Refine your solution multiple times based on verification results
- Conduct additional verification when the solution changes significantly
- Continue this cycle until you have a completely verified solution

CODE DOCUMENTATION REQUIREMENTS

All code blocks MUST include:

- A comment at the top explaining the purpose of the code
- Comments for each major section of code
- Descriptions of key variables and their purpose
- Explanation of any non-obvious logic or algorithms
- Clear documentation of expected inputs and outputs
- Comments explaining verification logic and what it confirms

LIBRARY USAGE GUIDELINES

When to Use NumPy

- For efficient numerical calculations and array operations
- For linear algebra operations (matrix multiplication, determinants, etc .)
- For generating random data in simulations
- For statistical functions and operations on large datasets
- For numerical integration and differentiation

When to Use SymPy

- For symbolic mathematics and algebraic manipulations
- For solving equations symbolically (rather than numerically)
- For symbolic differentiation and integration
- For series expansions and limits
- For simplifying algebraic expressions
- For working with fractions and exact values (rather than floating-point)
- For verifying mathematical identities and theorems

MATHEMATICAL NOTATION REQUIREMENTS

- ****Complete LaTeX Expressions****: Ensure all LaTeX expressions are properly closed and have matching delimiters
- ****Consistent Notation****: Use consistent variable naming throughout the solution
- ****Equation Numbering****: For multi-step solutions, number key equations for clear reference

- ****Proper Operators****: Use proper mathematical operators (x, /, in, forall, etc.) instead of plain text equivalents
- ****Formula Verification****: Double-check all formulas for typographical correctness
- ****Fractions****: Use proper LaTeX fraction notation ' $\frac{\text{numerator}}{\text{denominator}}$ ' for clarity
- ****Subscripts and Superscripts****: Use proper subscript and superscript notation
- ****Vector Notation****: Use consistent vector notation (bold, arrow, etc.)
- ****Set Notation****: Use proper set notation with curly braces and appropriate operators
- ****Alignment****: For multi-line equations, ensure proper alignment using appropriate LaTeX constructs

DOMAIN-SPECIFIC REQUIREMENTS

For Geometry Problems

- Use coordinate geometry in your code - this is MANDATORY for all geometry problems
- Use NumPy for vector and matrix operations when applicable
- Use SymPy for symbolic geometry when exact values are needed
- Calculate angles and distances numerically
- Verify geometric properties with explicit calculations
- Test your solution with multiple coordinate configurations
- Never rely on visual intuition - always verify with calculations
- For triangle problems, verify using both side lengths and angle measures
- For circles and polygons, verify all claimed relationships numerically
- Convert all geometric claims into algebraic equations and verify them

For Probability Problems

- Use simulation in your code to verify probability calculations - this is MANDATORY
- Use NumPy's random module for efficient simulation
- Generate a large number of trials (at least 10000)
- Compare analytical results with simulation results
- Verify that probability distributions sum to 1.0
- Use SymPy for exact probability calculations when appropriate
- Explicitly verify conditional probability calculations
- For combinatorial problems, verify with direct counting for small cases

- Always check that your counting includes all possible cases
- Test independence/dependence assumptions explicitly

For Algebraic Problems

- Use SymPy for symbolic manipulation and equation solving
- Test equations with numerical substitution in your code
- Verify all constraints are satisfied
- Check for extraneous solutions
- Test with multiple values to confirm patterns
- Verify polynomial roots by substitution
- For inequalities, check boundary cases and test points in each region
- Verify each algebraic manipulation step-by-step

HANDLING COMPLEX PROBLEMS

For high-complexity problems:

- Break the problem into smaller sub-problems
- Verify each sub-problem independently with code
- Track all assumptions explicitly
- Create a verification function for each major step
- Verify that sub-solutions combine correctly to solve the original problem
- Use concrete numerical examples to verify abstract reasoning
- For multi-variable problems, test with various combinations of values
- Use NumPy for numerical computations and SymPy for symbolic derivations

CASE ANALYSIS COMPLETENESS

For problems requiring case analysis:

1. Explicitly enumerate ALL possible cases
2. Verify each case independently with code
3. Confirm that your cases cover the entire solution space
4. Check for gaps in your case coverage
5. Verify boundary conditions between cases
6. Ensure edge cases are properly handled

CODE GUIDELINES

- Use Python for all code

- Use NumPy for numerical calculations and SymPy for symbolic mathematics
- Print intermediate values to make reasoning transparent
- Include comments explaining your code
- DO NOT attempt to create plots or visualizations
- Validate all calculations with code
- Use numpy, math, sympy, or other appropriate libraries as needed
- Implement verification functions that test your solution
- After each code output, explicitly interpret the results in relation to the problem
- Include error checking in your code when appropriate
- Use descriptive variable names that match the mathematical notation

HANDLING CONTRADICTIONS

If your verification reveals contradictions or inconsistencies:

1. ****STOP IMMEDIATELY**** and explicitly acknowledge the contradiction
2. Quote the specific verification output that revealed the contradiction
3. Identify which part of your reasoning or calculation contains the error
4. Explicitly state the implications of this contradiction for your solution
5. Revisit your assumptions and approach
6. Revise your solution completely, not just the final answer
7. Re-verify the revised solution with code
8. NEVER ignore contradictory evidence from verification

VARIABLE CONSISTENCY GUIDELINES

- ****Consistent Naming****: Once a variable is named, use exactly the same name and formatting throughout
- ****No Silent Substitutions****: When substituting one variable for another, explicitly state the substitution
- ****Clear Definitions****: When introducing important variables, clearly indicate what they represent
- ****Maintain Notation Systems****: If using a specific notation system (e.g ., vector notation), maintain it consistently
- ****Avoid Ambiguity****: Ensure your use of symbols cannot be misinterpreted throughout the solution

REASONING CLARITY REQUIREMENTS

B. Prompt Template for TIR Generation

- ****Clear Reasoning Flow****: Present your solution as a clear, uninterrupted chain of logical steps
- ****Explicit Transitions****: When changing approaches, explicitly state why the previous approach is being abandoned
- ****Restart Clarity****: If restarting with a new approach, mention this explicitly and explain why
- ****Forward References****: Avoid referencing calculations or conclusions you haven't yet established
- ****Reasoning Completeness****: Ensure each step of reasoning follows logically from previous steps
- ****Track Assumptions****: Explicitly list and track assumptions throughout your solution

Remember: CODE IS REQUIRED for both exploration and verification.
Solutions without substantial code will be considered incomplete. The solution process should be iterative, with willingness to revisit earlier stages when verification reveals issues.

Abbreviations

TUM	Technical University of Munich
AMC	American Mathematics Competitions
AIME	American Invitational Mathematics Examination
AIMO	AI Mathematical Olympiad
API	Application Programming Interface
AWQ	Activation-aware Weight Quantization
CE	Cross-Entropy
CoT	Chain of Thought
DPO	Direct Preference Optimization
GBFS	Greedy Best-First Search
GRPO	Group Relative Policy Optimization
GSM8K	Grade School Math 8K
JSON	JavaScript Object Notation
KTO	Kahneman-Tversky Optimization
LLM	Large Language Model
LoRA	Low-Rank Adaptation
MATH	Mathematics dataset by Hendrycks et al.
MCTS	Monte Carlo Tree Search
MDP	Markov Decision Process
OS	Oversampling
PPO	Proximal Policy Optimization
PRM	Process Reward Model
RL	Reinforcement Learning
RLHF	Reinforcement Learning from Human Feedback
SE	Standard Error
SOTA	State-of-the-Art
TIR	Tool-Integrated Reasoning
ToRA	Tool-integrated Reasoning Agent
VLLM	Versatile Large Language Model

List of Figures

3.1.	Failed transformation pipeline for converting pure mathematical solutions to tool-integrated reasoning traces. Stage 1 generates approaches with <i>predicted</i> code outputs, Stage 2 synthesizes them.	17
3.2.	Abridged illustrative tool-integrated solution to the octagon coloring problem using systematic enumeration to avoid analytical counting errors.	23
3.3.	Data reduction pipeline showing the progressive filtering from 40,000 generated traces through correctness verification and quality filtering to 3,410 final training examples. Each stage removes traces based on different criteria, with percentages showing the reduction relative to the previous stage.	24
3.4.	Impact of heuristic filtering on metric distributions. Each panel shows the distribution of a key metric before (gray) and after (blue) filtering, with red dashed lines indicating the filtering thresholds. The significant shifts in distributions demonstrate how filtering selects for traces with balanced code-to-text ratios, appropriate lengths, minimal comment overhead, and continuous computational engagement.	27
4.1.	State diagram of the TIR multi-turn conversation orchestration process. The system alternates between natural language reasoning and code execution in a persistent computational dialogue.	44
4.2.	Training dynamics for LoRA fine-tuning of Qwen3-8B showing (top) training loss with best validation checkpoint marked at step 1536, (middle) gradient norm evolution demonstrating stable optimization, and (bottom) learning rate schedule over 10 epochs. The consistent loss reduction and stable gradient norms indicate healthy training dynamics without optimization difficulties.	49
5.1.	PRM training dynamics showing loss progression and learning rate schedule over the single training epoch. Despite notable variance in the training loss, the overall trend shows steady improvement of the implicit reward model from ~ 0.7 to ~ 0.27	56

- 6.1. Scaling behavior of majority voting versus GBFS+PRM as search intensity increases. Search intensity represents n (number of independent attempts) for majority voting and b (branching factor) for GBFS+PRM. The dotted line shows the single-attempt direct solving baseline (21.56%). 62

List of Tables

3.1. Source datasets for tool-integrated reasoning generation. DeepScaler provides breadth through comprehensive coverage, while LIMO offers depth through extreme-difficulty problems.	19
3.2. Performance comparison of Gemini 2.0 Flash with and without the code-first TIR prompt on AIME 2024 problems.	20
3.3. Heuristic filtering thresholds applied to ensure structural quality of tool-integrated reasoning traces	28
3.4. Behavior filtering criteria and their impact on dataset composition. All thresholds must be met simultaneously for trace retention.	33
3.5. Final dataset composition after multi-stage quality filtering	40
4.1. Base Model Distribution	47
4.2. Top 5 AIME 2024 Performing Configurations	48
4.3. AWQ Quantization Performance Analysis	51
6.1. AIME 2025 performance comparison across evaluation modes with 32x oversampling. The improvement column shows absolute percentage point gains over the direct solving baseline.	61

Bibliography

- [Cob+21] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, C. Hesse, and J. Schulman. *Training Verifiers to Solve Math Word Problems*. 2021. arXiv: 2110.14168 [cs.LG].
- [Dee25] DeepSeek-AI. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. 2025. arXiv: 2501.12948 [cs.CL].
- [Det+23] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer. *QLoRA: Efficient Finetuning of Quantized LLMs*. 2023. arXiv: 2305.14314 [cs.LG].
- [DZ22] T. Dettmers and L. Zettlemoyer. *The case for 4-bit precision: k-bit Inference Scaling Laws*. 2022. arXiv: 2212.09720 [cs.LG].
- [Fra+23] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh. *GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers*. 2023. arXiv: 2210.17323 [cs.LG].
- [Gan+25] K. Gandhi, A. Chakravarthy, A. Singh, N. Lile, and N. D. Goodman. *Cognitive Behaviors that Enable Self-Improving Reasoners, or, Four Habits of Highly Effective STaRs*. 2025. arXiv: 2503.01307 [cs.CL].
- [Gao+24] B. Gao, F. Song, Z. Yang, Z. Cai, Y. Miao, Q. Dong, L. Li, C. Ma, L. Chen, R. Xu, Z. Tang, B. Wang, D. Zan, B. Chang, T. Liu, and M. Lin. *Omni-MATH: A Universal Olympiad Level Mathematic Benchmark For Large Language Models*. 2024. arXiv: 2410.07985 [cs.CL].
- [Gou+24] Z. Gou, Z. Shao, Y. Gong, Y. Shen, Y. Yang, M. Huang, N. Duan, and W. Chen. *ToRA: A Tool-Integrated Reasoning Agent for Mathematical Problem Solving*. 2024. arXiv: 2309.17452 [cs.CL].
- [Gra+24] A. Grattafiori, A. Dubey, A. Jauhri, et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI].
- [Hen+21] D. Hendrycks, C. Burns, S. Kadavath, A. Arora, S. Basart, E. Tang, D. Song, and J. Steinhardt. *Measuring Mathematical Problem Solving With the MATH Dataset*. 2021. arXiv: 2103.03874 [cs.LG].

- [Hu+21] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: 2106.09685 [cs.CL].
- [Kap+20] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. *Scaling Laws for Neural Language Models*. 2020. arXiv: 2001.08361 [cs.LG].
- [Lew+22] A. Lewkowycz, A. Andreassen, D. Dohan, E. Dyer, H. Michalewski, V. Ramasesh, A. Slone, C. Anil, I. Schlag, T. Gutman-Solo, Y. Wu, B. Neyshabur, G. Gur-Ari, and V. Misra. *Solving Quantitative Reasoning Problems with Language Models*. 2022. arXiv: 2206.14858 [cs.CL].
- [Lig+23] H. Lightman, V. Kosaraju, Y. Burda, H. Edwards, B. Baker, T. Lee, J. Leike, J. Schulman, I. Sutskever, and K. Cobbe. *Let’s Verify Step by Step*. 2023. arXiv: 2305.20050 [cs.LG].
- [Lin+24] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han. *AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration*. Originally submitted in 2023, updated version 5 from July 2024. 2024. arXiv: 2306.00978 [cs.CL].
- [Luo+25] M. Luo, S. Tan, J. Wong, X. Shi, W. Y. Tang, M. Roongta, C. Cai, J. Luo, L. E. Li, R. A. Popa, and I. Stoica. *DeepScaleR: Surpassing O1-Preview with a 1.5B Model by Scaling RL*. <https://pretty-radio-b75.notion.site/DeepScaleR-Surpassing-O1-Preview-with-a-1-5B-Model-by-Scaling-RL-19681902c1468005bed8ca303013a4e2>. Notion Blog. 2025.
- [Mos+25] I. Moshkov, D. Hanley, I. Sorokin, S. Toshniwal, C. Henkel, B. Schifferer, W. Du, and I. Gitman. *AIMO-2 Winning Solution: Building State-of-the-Art Mathematical Reasoning Models with OpenMathReasoning dataset*. 2025. arXiv: 2504.16891 [cs.AI].
- [RUC25] RUCAIBOX STILL Team. *STILL-3-Preview-RL-Data*. <https://huggingface.co/datasets/RUC-AIBOX/STILL-3-Preview-RL-Data>. Hugging Face dataset of approximately 30K math QA pairs from MATH, NuminaMath-CoT, and AIME 1983–2023. 2025.
- [Tea25] Q. Team. *Qwen3 Technical Report*. 2025. arXiv: 2505.09388 [cs.CL].
- [Wan+24] P. Wang, L. Li, Z. Shao, R. X. Xu, D. Dai, Y. Li, D. Chen, Y. Wu, and Z. Sui. *Math-Shepherd: Verify and Reinforce LLMs Step-by-step without Human Annotations*. 2024. arXiv: 2312.08935 [cs.AI].
- [Ye+25] Y. Ye, Z. Huang, Y. Xiao, E. Chern, S. Xia, and P. Liu. *LIMO: Less is More for Reasoning*. 2025. arXiv: 2502.03387 [cs.CL].

- [Yua+24] L. Yuan, W. Li, H. Chen, G. Cui, N. Ding, K. Zhang, B. Zhou, Z. Liu, and H. Peng. *Free Process Rewards without Process Labels*. 2024. arXiv: 2412.01981 [cs.LG].