Computational Thinking Reasoning in Large Language Models

Anonymous ACL submission

Abstract

While large language models (LLMs) have demonstrated remarkable reasoning capabilities, they often struggle with complex tasks that require specific thinking paradigms, such as divide-and-conquer and procedural deduction, *etc*. Previous researches integrate external, reliable tools to alleviate logical inconsistencies and hallucinations in LLMs' problem-solving processes. However, we argue that the root challenge is more profound: LLMs lack the complex thinking paradigms (*i.e*., computational thinking) during reasoning. In this paper, we propose Computational Thinking Model (CTM), a novel framework that incorporates computational thinking paradigms into LLMs. This framework enables LLMs to reformulate complex problems through decomposition, abstraction, reduction, and simulation, among other techniques. Specifically, live code execution is seamlessly integrated into the reasoning process, allowing CTM to think by computing. CTM directly instills computational thinking objectives into LLMs through tailored reinforcement learning rewards, which encourages problem simplification, modular planning, and iterative verification. We conduct extensive evaluations on multiple code generation and mathematical benchmarks. The results demonstrate that CTM outperforms conventional reasoning models and tool-augmented baselines in terms of accuracy, interpretability, and generalizability. We hope this study offers valuable insights for AI reasoning, where LLMs can transform problems into robust, verifiable, and scalable computational workflows, much like computer scientists do.

1 Introduction

Recent advancements in large language models (LLMs), particularly those enhanced through reinforcement learning (RL), have significantly pushed the boundaries of what these models can achieve in natural language reasoning (OpenAI, 2025). Models such as DeepSeek-R1 (Guo et al., 2025) have demonstrated remarkable capabilities. However, in complex domains like algorithmic problem-solving and mathematical reasoning, these models often face fundamental limitations when dealing with structured and symbolic reasoning tasks, where precision, consistency, and compositionality are essential. In these contexts, errors in intermediate steps often propagate through the reasoning pipeline unchecked, making it challenging for the models to produce reliable solutions (Lightman et al., 2023; Chen et al., 2022; Hendrycks et al., 2021; Jain et al., 2024; Zhang et al., 2024b,a, 2025a).

A critical limitation arises from the fact that most natural language reasoning models perform reasoning as a purely generative process. While advanced chain-of-thought prompting enables the articulation of intermediate reasoning steps, it does not provide the model with mechanisms to systematically verify its outputs or revise erroneous intermediate conclusions. Some current researches alleviate these issues by supplementing the reasoning capabilities of LLMs with external and reliable tools, such as web search and code interpreter (Schick et al., 2023; Feng et al., 2025; Qian et al., 2025; Li et al., 2025b; Wang et al., 2024). Although these augmentations are beneficial, LLMs often treat reasoning as an opaque generation process. In this context, the usage of tools serves as auxiliary support rather than being integrated into a cohesive thinking paradigm. To address these challenges, we contend that the fundamental limitation is not merely the lack of tools, but rather the absence of essential thinking paradigms that underpin the reasoning process. For computer scientists, foundational methodologies such as decomposition, reduction, abstraction, and simulation form the core of computational thinking (Wing, 2006), enabling them to tackle complex problems effectively.

In general, computational thinking is a cognitive paradigm that involves transforming complex prob-

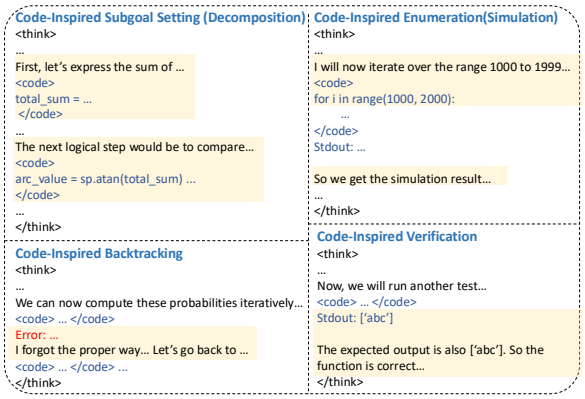


Figure 1: Computational thinking with code inspired patterns from our model.

lems into more manageable forms through methodologies such as decomposition, enumeration and verification. *E.g.*, those who have mastered computational thinking often ask critical questions during their problem-solving process, including but not limited to following questions (Wing, 2006): Can the given task be broken down into modular subgoals? Can current optimal state be derived from previous states? The thinking paradigm underlying these questions fosters deep and systematic reasoning, enabling more effective problem-solving.

Drawing inspiration from this principle, we propose Computational Thinking Model (abbreviated as CTM), a framework that enables LLMs to integrate computational thinking as a primary reasoning strategy. Specifically, CTM provides an interactive execution environment where the reasoning process of LLMs is enhanced through live code execution. The multi-turn interaction facilitates decomposition of monolithic reasoning tasks, abstraction beyond textual limitations, and explicit intermediate computation in place of implicit speculation. CTM allows LLMs to learn computational thinking within this environment through reinforcement learning (RL), with supervised fine-tuning (SFT) serving as the cold start warmup. LLMs acquire specific thinking methodologies from our constructed dataset during the SFT phase, and they further expand these capabilities during the RL phase. The resulting model exhibits behavior that is not only more accurate, but also systematically grounded in executable logic. As illustrated in Figure 1, the incorporation of code-inspired computational thinking enables the model to solve complex reasoning tasks with great consistency, transparency, and precision.

We conduct extensive experiments on code and mathematical benchmarks (Chen et al., 2021; Austin et al., 2021; Jain et al., 2024; Li et al., 2022; MAA, 2025; Lightman et al., 2023). Experimental results suggest that CTM outperforms conventional reasoning LLMs as well as tool-augmented baselines (Guo et al., 2025; OpenAI, 2025; Ahmad et al., 2025; Claude, 2024; Team, 2025; Yang et al., 2024; Wang et al., 2024; Labs, 2025; Thoughts, 2025; Open-R1, 2025). CTM not only introduces a new technique but also offers a novel perspective on how LLMs can reason more effectively. By adopting the thinking paradigms of computer scientists, LLMs are better equipped to solve problems that require compositionality and precision. The framework of CTM holds great potential to make a broad impact on future intelligent systems.

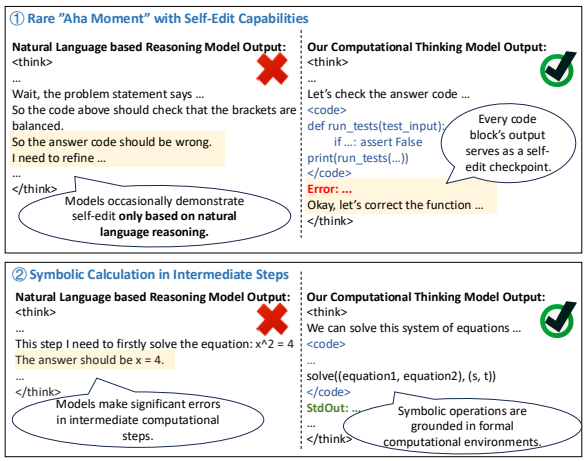


Figure 2: Limitation cases in natural language-based reasoning models. (Cases in the left are from DeepSeek-R1, one of the most powerful reasoning models.)

2 Motivating Examples

We present a motivating example in Figure 2. Despite their great performance under code generation and mathematics, existing LLMs still struggle in several key points that affect the reasoning.

2.1 Limitations in Natural Language-based Reasoning

The Rare "Aha Moment" with Self-Edit Capabilities. Current language models demonstrate an intriguing phenomenon in their reasoning behavior. While capable of generating multi-step solutions through chain-of-thought prompting, they typically produce linear reasoning paths with limited capacity for self-correction. This creates a fundamental limitation where errors in early reasoning steps propagate unchecked through subsequent stages, as the models lack robust mechanisms to identify and revise flawed intermediate conclusions.

What makes this particularly interesting is that when models do occasionally demonstrate effective self-edit during training, researchers often describe it as an "aha moment" (see the left upper part of Figure 2). These moments are noteworthy not because they represent paradoxical behavior, but because they reveal the model's latent potential for self-editing that is otherwise difficult to elicit. The rarity of these occurrences highlights how challenging it is to train models to consistently recognize and correct their own reasoning errors.

Human problem-solving naturally incorporates continuous self-monitoring and adjustment-a capability that remains largely absent in current AI systems. While reinforcement learning approaches have made progress by introducing external feedback loops, the models still struggle with intrinsic self-correction. The celebrated "aha moments" represent promising but isolated cases where models overcome this limitation, suggesting more systematic approaches are needed to make self-correction a reliable rather than serendipitous capability.

This gap becomes particularly apparent in complex reasoning tasks, where the ability to dynamically revise one's approach could significantly improve performance. The current situation presents both a challenge and an opportunity: while spontaneous self-correction remains rare, its occasional occurrence proves that models do possess this capability in principle, waiting to be properly harnessed through improved training paradigms.

Symbolic Calculation in Intermediate Steps. Current LLMs approach symbolic operations (*e.g.* algebraic manipulation, logical inference) through textual pattern matching rather than structured computation, as discussed in prior work (Hu et al., 2024). While these models can approximate symbolic transformations using statistical correlations from their training data, they frequently violate fundamental mathematical rules and semantic constraints. This limitation manifests clearly in tasks requiring rigorous symbolic manipulation, such as polynomial factorization or modular arithmetic, where models often generate syntactically plausible but mathematically invalid derivations.

The core issue stems from LLMs treating mathematical symbols as surface-level tokens rather than as manipulable objects governed by formal systems. Our analysis of the powerful DeepSeek-R1 model reveals this fundamental weakness: when presented with complex calculation tasks, the model not only produces incorrect final answers but also makes significant errors in intermediate computational steps (see the left bottom part of Figure 2). Notably, these errors occur even for simple calculations. This observation highlights the inherent risks of relying solely on natural language-based reasoning for symbolic computation, regardless of problem complexity. The persistent failure to maintain mathematical validity across both simple and complex operations suggests a fundamental limitation in the current paradigm's ability to support rigorous symbolic reasoning.

2.2 Computational Thinking as a Foundational Remedy

As (Wing, 2006) says, *computational thinking involves solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science.* Our Computational Thinking Model (CTM) addresses these limitations by reconceptualizing original occasional reasoning as an iterative, executable process. By interleaving natural language with live code execution, CTM introduces execution feedback loops that enable dynamic self-edit—every code block's output serves as a validation checkpoint in Figure 2. Symbolic operations are grounded in formal computational environments, ensuring rule-based rigor. This paradigm shift from generative text construction to computational workflow execution yields more reliable, scalable, and verifiable problem-solving. It helps our CTM show various complex code inspired reasoning behaviors, as shown in Figure 1.

3 Method

We propose the **Computational Thinking Model (CTM)**, an interactive reasoning framework that interleaves natural language analysis and code-based execution to tackle challenging tasks. In our computational thinking model, large language models alternate between planning/reflection and executable code, thereby grounding intermediate reasoning in concrete computations. In this section, we will describe our framework architecture, the training approach, and the inference procedure supported by real-time code execution in detail.

3.1 Language-Code Reasoning Framework

The core idea is to let the model "think by computing." Figure 3 illustrates how the model iterates

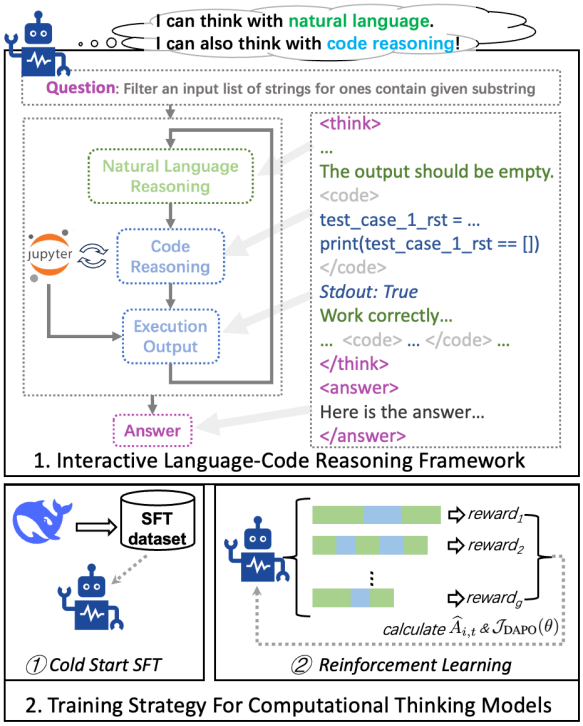


Figure 3: The illustration of the framework and training strategy for our computational thinking model.

between natural language reasoning, code reasoning, and execution feedback.

Code Execution Integration. Starting from an initial input question or prompt, the Computational Thinking Model (CTM) iteratively appends each newly generated natural language reasoning or code snippet to the shared reasoning context. Whenever the model emits a code block enclosed within `<code>...</code>` tags, the system extracts the enclosed code snippet and executes it in a sandbox environment with persistent memory—functionally similar to the notebook-style execution used in environments like Jupyter. This execution enables the model to iteratively validate hypotheses, perform computations, and refine intermediate steps, much like a human programmer leveraging an interactive coding platform to test and debug solutions during problem-solving tasks.

The results of code execution, including standard outputs (`stdout`), error messages (*e.g.* syntax or runtime errors), or returned values, are automatically captured and transformed into text. These textual outputs are subsequently appended to the model's reasoning context, serving as feedback for the next iteration. This structured feedback loop not only grounds the model's reasoning in verifiable computation but also enables real-time debugging and refinement, facilitating dynamic problem-solving. The iterative process continues until the model generates an explicit `</think>` tag, signaling the completion of the reasoning process and providing the final solution.

This loop mirrors the behavior of human programmers who incrementally solve problems by writing, executing, and refining code within interactive computing environments. It allows CTM to model the problem-solving process with a higher degree of flexibility, accuracy, and interpretability. The underlying process is outlined in Algorithm 1 in the Appendix. The sandbox environment is particularly crucial, as it supports persistent memory across code executions, allowing variable states, functions, and intermediate results to persist across iterations. This feature enables the model to build upon its own computations dynamically, reflecting the iterative and modular nature of computational thinking. By embedding computational feedback directly into the reasoning process, CTM transcends traditional static reasoning paradigms and achieves a level of adaptability and transparency akin to human problem-solving strategies.

Execution Environment and Result Integration. We implement a notebook-style *execution environment* that persists state across multiple code cells. This environment can capture syntax or runtime errors as textual feedback, records standard outputs, and makes newly defined variables or functions accessible to subsequent code blocks. Hence, real-time *compute-and-check* stages guide the model's reasoning by enabling dynamic refinement: if a mistake arises in part of the solution, the model can revise its code and retest until reaching a sufficiently correct final answer. Implementation details are shown in Appendix F.

3.2 Training Strategy

To effectively incorporate computational thinking principles into our model's reasoning capabilities, we employ a two-phase training strategy designed to align the model's reasoning with natural language and code execution. These phases consist of: (1) an initial supervised fine-tuning ("cold start") phase to familiarize the model with natural language and code-mixed reasoning traces, and (2) a reinforcement learning (RL) phase leveraging code-based rollouts and Decouple Clip and Dynamic Sampling Policy Optimization (DAPO) (Yu et al., 2025), an advanced policy optimization algorithm inspired by GRPO (Shao et al., 2024). Below, we describe each phase in detail.

Cold Start Supervised Fine-tuning. The supervised fine-tuning (SFT) phase aims to establish the foundational reasoning patterns required for integrating computational thinking into the model. To achieve this, we trained the model on curated datasets comprising reasoning tasks annotated with mixed natural language and executable code traces. These traces are designed to interleave natural language explanations and code snippets while reflecting the iterative problem-solving process characteristic of computational thinking.

The training datasets include a diverse range of tasks drawn from both mathematical reasoning and programming challenges. We sample 13k code questions and 11.9k math questions from existing open-sourced dataset (Ahmad et al., 2025; inclusionAI, 2025). We use the few-shot prompt to guide advanced language models like DeepSeek-V3 (Liu et al., 2024) to generate natural language and code mixed solution traces. Each solution trace consists of a sequence of structured reasoning steps, including hypothesizing, implementation attempts, debugging, refinement, and validation. For example, a typical trace might begin with a high-level hypothesis, followed by a naive implementation, iterative corrections based on execution feedback, and finally an optimized solution validated against the problem constraints. The training corpus is designed to encourage the model to adopt structured reasoning patterns that align closely with computational thinking. These patterns may involve strategies like divide-and-conquer decomposition, dynamic programming, heuristic search, or brute-force refinement. In practice, we construct some few-shot examples in these patterns to help the dataset construction. For instance, a prototypical trace might follow the workflow: `understand` → `plan` → `code` → `validate` → `refine` → `finalize`. By exposing the model to such structured reasoning patterns, the fine-tuning phase effectively establishes computational thinking concepts—including decomposition, abstraction, iteration, simulation, and transformation—as integral components of the reasoning process.

During SFT training, we mask out execution outputs from the sandbox environment (*e.g.* runtime results or error messages). This ensures that the model does not rely on predicting specific values or computational outputs directly but instead uses reasoning patterns to arrive at consistent and generalizable solutions. Such masking prevents over-reliance on data memorization and significantly improves training stability by guiding the model to focus on logical patterns rather than specific numerical outcomes. For instance, instead of inferring execution outputs in purely natural language contexts, the model learns to scaffold its reasoning around the structural process of problem-solving. Furthermore, SFT encourages the model to leverage computational feedback during intermediate stages, laying the foundation for dynamic refinement and self-correction. This flexibility enables the model to adapt to diverse problem-solving scenarios without being constrained by rigid reasoning templates, making it highly versatile across tasks requiring structured yet adaptive reasoning.

Reinforcement Learning with NL-Code-Mixed Rollouts. Following the supervised fine-tuning phase, we further align the model's behavior by employing reinforcement learning (RL). This phase optimizes the model's reasoning efficiency and reliability across both natural language and executable code. By generating dynamically mixed rollouts of natural language and code snippets, the model develops emergent strategies characteristic of computational thinking, further refining its ability to reason iteratively and adaptively.

To ensure stable optimization in the presence of mixed modality outputs, we adopt the DAPO algorithm (Yu et al., 2025), a tailored variant of GRPO. The RL training process begins by sampling multiple rollouts for each task and evaluating them against a rule-based reward function. We sample 2k code questions and 2k math questions from open-sourced dataset (Skywork, 2025).

Given a question q with a ground-truth answer a, the model samples G reasoning trajectories $\{o_i\}_{i=1}^{G}$ from the current policy $\pi_{\theta_{\text{old}}}$, where each trajectory interleaves natural language reasoning steps and executable `<code>...</code>` blocks within the shared sandbox environment. To provide supervision, each trajectory is assigned a scalar reward R_i, reflecting the quality of its final output and intermediate reasoning. Rewards are computed via domain-specific evaluation tools as follows: **Code Question Judgments:** Correct solutions that pass all test cases within the sandbox environment *Sandbox Fusion* (Liu, 2024) are assigned a reward of $+1$, while solutions that fail the test cases receive -1. **Mathematical Question Judgments:** Accurate solutions using the *math-verify* tool receive a reward of $+1$, while incorrect solutions are assigned -1.

To align the model's policy π_θ with computational thinking principles, DAPO maximizes the normalized token-level advantage while constraining updates using asymmetric clipping to ensure stable learning. For each sampled question-answer pair (q, a) and its associated rollouts $\{o_i\}_{i=1}^{G}$, the optimization objective is defined as:

$$\mathcal{J}_{\text{DAPO}}(\theta) = \mathbb{E}_{(q,a),\{o_i\}} \left[\frac{1}{\sum_{i=1}^{G} |o_i|} \sum_{i=1}^{G} \sum_{t=1}^{|o_i|} \right. \quad (1)$$

$$\left. \min\left(r_{i,t}(\theta)\cdot\widehat{A}_{i,t}, \text{clip}\left(r_{i,t}(\theta), 1-\epsilon_{\text{l}}, 1+\epsilon_{\text{h}}\right)\cdot\widehat{A}_{i,t}\right) \right].$$

where $r_{i,t}(\theta) = \frac{\pi_\theta(o_{i,t} \mid q, o_{i,<t})}{\pi_{\text{old}}(o_{i,t} \mid q, o_{i,<t})}$ and $\widehat{A}_{i,t} = \frac{R_i - \mu_R}{\sigma_R}$. Only batches containing both successful and failed rollouts (*i.e.*, $0 < \#\{\text{correct } o_i\} < G$) are used, ensuring a meaningful gradient signal. This training approach drives the model to iteratively refine complex solutions by writing code, evaluating the results, and adjusting generation patterns accordingly, reinforcing the "computational thinking" principle.

3.3 Inference with Code Execution

At inference time, CTM applies the same language-code reasoning loop as during training. Given an input query, the model alternates between generating natural language reasoning steps and executable code snippets. Each code block is executed in real-time within a sandbox environment, and the generated output—whether standard print responses, errors, or returned values—is appended to the reasoning history. The model dynamically adjusts its reasoning trajectory in response to execution outcomes, iteratively refining its solution until a final answer is produced. This iterative refinement ensures that errors detected during intermediate steps can be corrected, reducing the likelihood of logical or computational failures.

Compared to original natural language based reasoning models, CTM's iterative language-code cycle enhances robustness and interpretability, as each step in the reasoning process is grounded by verifiable computations. This approach not only facilitates accurate problem-solving but also provides a transparent trace of the reasoning process, making CTM a versatile framework for tasks requiring multi-step reasoning and computational rigor.

4 Study Design

We evaluate CTM on nine benchmarks that cover a broad spectrum of reasoning challenges, encompassing both code and non-code domains. Functional code generation benchmarks contain HumanEval, HumanEval+, MBPP, and MBPP+ (Chen et al., 2021; Austin et al., 2021; Liu et al., 2023a), focusing on functional correctness in code generation with extended test cases. Programming contest benchmarks include CodeContests and LiveCodebench (Li et al., 2022; Jain et al., 2024). We further apply three mathematical benchmarks, including AIME 2024 and AIME 2025 benchmarks (MAA, 2025), MATH-500 (Lightman et al., 2023). Detailed benchmark descriptions and evaluation metrics are provided in Appendix D.

We evaluate against three categories of state-of-the-art reasoning systems, including proprietary source models (containing Anthropic's `Claude-3.5-Sonnet` (Claude, 2024), `OpenAI's GPT-4o-0513` (OpenAI, 2024), and `o1` series models (OpenAI, 2025)), open-source models (`Bespoke-Stratos-32B` (Labs, 2025), `OpenThinker-32B` (Thoughts, 2025), `OlympicCoder-32B` (Open-R1, 2025), and `OCR-Qwen-32B` (Ahmad et al., 2025), along with `DeepSeek-V3` (Liu et al., 2024) and `R1` (Guo et al., 2025)), and tool-augmented agents (`OpenHands` (Wang et al., 2024)). Considering that CTM is continuously trained from `Qwen2.5-32B-Instruct`, we evaluate the performance of `Qwen2.5-32B-Instruct` (Yang et al., 2024) to further analyze the effectiveness of our CTM.

5 Experimental Results

Performance on Code Problems. Tables 1 and 2 summarize the performance of our model against state-of-the-art large reasoning models, code-specialized models, and agent frameworks on code-related benchmarks. For function-level code generation tasks such as HumanEval(+) and MBPP(+), our model gains advantages from the code execution and refinement loop in our natural language and code-mixed reasoning traces. It can autonomously learn to correct errors that occur in the code and provide the final answer, as shown in Figures 2 and 5. For complex code competition problems, our model demonstrates superior performance on both LiveCodeBench and CodeContests, two challenging benchmarks evaluating program

Table 1: Code Generation Performance on Complex Competitive Programming Tasks

Model	LiveCodeBench	CodeContests
Proprietary Source Models		
Claude-3.5-Sonnet-1022	38.9	-
GPT-4o-0513	32.9	-
OpenAI o1-mini	53.8	-
OpenAI o1-1217	63.4	-
Open-Source Models		
DeepSeek-V3-671B	36.2	20.2
DeepSeek-R1-671B	65.9	26.2
QwQ-32B	61.3	20.2
Bespoke-Stratos-32B	30.1	6.3
OpenThinker-32B	54.1	16.4
OlympicCoder-32B	57.4	18.0
OCR-Qwen-32B	61.8	24.6
Tool-augmented Agent Frameworks		
OpenHands (DeepSeek-V3-671B)	39.3	23.9
Our Base Model		
Qwen2.5-32B-ins	33.4	11.1
Our Method		
CTM-32B-SFT	61.6	24.2
CTM-32B-RL	**66.5**	**28.5**

Table 2: Code Generation Performance on Function-level Code Generation Tasks

Model	HumanEval	H+	MBPP	M+
Proprietary Source Models				
Claude-3.5-Sonnet-1023	92.1	85.4	86.0	72.2
GPT-4o-0513	91.5	87.8	89.4	74.6
Open-Source Models				
DeepSeek-V3-671B	94.5	89.0	90.2	74.9
Bespoke-Stratos-32B	88.4	84.1	87.6	74.1
OpenThinker-32B	88.4	85.4	89.4	75.4
OlympicCoder-32B	82.3	75.0	83.6	72.0
OCR-Qwen-32B	81.1	77.4	82.8	68.8
Tool-augmented Agent Frameworks				
OpenHands (DS-V3-671B)	**95.5**	90.5	91.0	76.6
Our Base Model				
Qwen2.5-32B-ins	88.4	83.7	84.0	74.3
Our Method				
CTM-32B-SFT	91.3	85.0	90.2	78.3
CTM-32B-RL	**93.7**	**92.2**	**92.5**	**79.1**

reasoning and generation. Specifically, our method with reinforcement learning (RL) achieves the best results, scoring 66.5 on LiveCodeBench and 28.5 on CodeContests, outperforming other leading reasoning models such as `DeepSeek-R1` and code-specialized models like `OCR-Qwen-32B`. Compared to `OpenAI o1-mini` and `DeepSeek R1`, which are widely regarded as the state-of-the-art large reasoning models, our approach achieves significant gains with its interactive natural language and code mixed reasoning capabilities.

Performance on Math Problems. Our model also achieves top-tier performance across those general domain tasks such as mathematical, as shown in Table 3. Our RL-trained variant achieves results superior to or comparable with those of powerful proprietary and open-source models. Specifically, our RL-trained model achieves scores of 76.6 on AIME 2024 and 66.6 on AIME 2025—complex competitive math benchmarks—and reaches 96.8 on the MATH-500 dataset.

Table 3: Math Performance Comparison with State-of-the-Art Models

Model	AIME 2024	AIME 2025	MATH-500
Proprietary Source Models			
Claude-3.5-Sonnet-1022	16.0	13.3	78.3
GPT-4o-0513	9.3	6.6	74.6
OpenAI o1-mini	63.6	60.0	90.0
OpenAI o1-1217	79.2	73.3	96.4
Open-Source Models			
DeepSeek-V3-671B	39.2	36.6	90.2
DeepSeek-R1-671B	70.0	**76.6**	**97.3**
QwQ-32B	**79.5**	69.5	96.0
Bespoke-Stratos-32B	30.0	23.3	93.0
OpenThinker-32B	33.3	23.3	90.6
OlympicCoder-32B	40.0	36.6	94.0
OCR-Qwen-32B	36.6	30.0	92.2
Tool-augmented Agent Frameworks			
OpenHands (DS-V3-671B)	50.0	46.6	93.6
Our Base Model			
Qwen2.5-32B-ins	6.6	6.6	86.6
Our Method			
CTM-32B-SFT	70.0	56.6	92.2
CTM-32B-RL	**76.6**	**66.6**	**96.8**

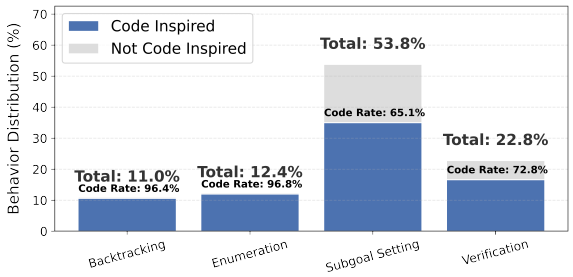


Figure 4: Comparison of Reasoning Behavior Distributions for our computational thinking model. Case studies for each behavior are shown in Figure 1.

To measure the contribution of our proposed training methodology, we further analyze the performance improvement for difference stages in our training process, relative to our base model without computational thinking integration in Table 4. Our fine-tuned model (`SFT`) achieves substantial improvements across all benchmarks, highlighting the impact of task-specific supervised fine-tuning with computational thinking-flavored data. Training with reinforcement learning further improves performance. Notably, it provides a gain of *+33.1* on LiveCodeBench and *+70.0* on AIME 2024.

5.1 Analysis of Reasoning Behavior

To investigate how the added code execution capability enhances the computational thinking model's ability to solve complex problems and affect the reasoning behaviors, we analyze the reasoning trajectories of CTM. Following prior work on reasoning behavior analysis (Gandhi et al., 2025; Zeng et al., 2025), we leverage `GPT-4o` to annotate

reasoning-related behaviors, including *Backtracking*, *Verification*, *Subgoal Setting*, and *Enumeration*. In addition, we introduce a new annotation criterion to determine whether each behavior is influenced by code execution—resulting in variants such as *Code-Inspired Backtracking*, *Code-Inspired Verification*, etc. For each reasoning output, the annotation model is instructed to assign only one primary behavior category. The distribution of the behaviors for our computational thinking model is shown in Figure 4 on our sampled 500 questions from (Ye et al., 2025). Case studies for each behavior of our model are shown in Figure 1.

The results reveal that while the overall distribution across the four reasoning behavior categories is relatively similar between our model and other reasoning models (Gandhi et al., 2025; Zeng et al., 2025), a key distinction emerges in the degree of *code inspiration*. Within each behavior type, our model demonstrates a significantly higher proportion of code-inspired reasoning, suggesting that the integrated code execution mechanism effectively guides the model toward more accurate and grounded forms of reasoning. In contrast, in cases marked as *Verification*, `DeepSeek-R1` can only rely on natural language to evaluate intermediate results—an approach that introduces risks of hallucination and limitations in Section §2.

We also show the reinforcement learning training curves of our computational thinking model training process in Figure 6. The reward curve demonstrates continuous improvement in the model's overall performance during RL. The model rapidly learns to leverage code execution as an effective strategy for handling intermediate reasoning steps. We further show the case study in §A and Figure 5.

5.2 Ablation Study

We perform an ablation study in Table 4 to investigate the importance of code integration within our framework. Specifically, we first examine the effect of removing code integration during the training phase, where the model is trained to reason solely using natural language—following the common design of existing language-only reasoning models. In this setting, we retain the same training questions in both the SFT and RL stages as in our original experimental setup. We conduct a separate ablation in which code integration is disabled during the inference stage. In this case, our CTM model is used to produce outputs without invoking the code execution environment, effectively forcing it to rely exclusively on natural language reasoning. The results indicate that code integration plays a critical role in both the training and inference phases.

Table 4: Performance Improvements and Ablation Studies

Model Variant	Code		Math (AIME)	
	LiveCode	Contest	2024	2025
Qwen2.5-32B-ins	33.4	11.1	6.6	6.6
Our CTM				
SFT	61.6	24.2	70.0	56.6
SFT + RL	**66.5**	**28.5**	**76.6**	**66.6**
Training w/o code				
SFT	61.1	25.2	70.0	53.3
SFT + RL	63.5	26.9	76.6	60.0
Inference w/o code				
SFT	56.3	22.1	63.3	56.6
SFT + RL	58.5	23.8	63.3	53.3

6 Related Work

Building upon the concept of human tool use, recent efforts focus on augmenting LLMs with external computational tools to tackle inherently complex tasks. Tool-integrated reasoning, first introduced to address computationally intensive requirements in domains such as mathematics (Chen et al., 2022; Schick et al., 2023; Yao et al., 2023; Zhang et al., 2023b,a; Li et al., 2025a; Ma et al., 2025; Paranjape et al., 2023; Lu et al., 2023), provides LLMs access to environments for feedback.

In recent concurrent work, reinforcement learning (RL) has emerged as a powerful approach for training tool-integrated reasoning models (Feng et al., 2025; Qian et al., 2025; Li et al., 2025b; Wang et al., 2024). For instance, the ReTool framework (Feng et al., 2025) applies RL to optimize code interpreter usage specifically for mathematical reasoning tasks, and observes relatively simple tool-use patterns, such as basic code self-correction. We further analyze the research path for these important baselines in Appendix H.

7 Conclusion

We propose the computational thinking model (CTM), a novel reasoning model designed to integrate computational thinking principles into large language models. CTM provides an interactive reasoning process, making the large reasoning model not just generate text, but think like computer scientists—transforming problems into computational workflows that are robust and verifiable.

Limitations

Our results may be affected by the model training configuration and the design of the execution environment. The curated reasoning-code traces used for supervised fine-tuning could introduce bias if they overrepresent certain problem types or tool usage patterns. We conduct ablation studies to evaluate core components (*e.g.* the code integration for training and inference) in §5.2.

Our evaluation is based on benchmarks such as LiveCodeBench, CodeContests, and AIME. These cover diverse reasoning scenarios, but may not fully capture the complexity of real-world tasks involving long-term planning or multi-modal inputs. Moreover, our computational thinking model currently supports only Python-based code execution, which may limit generalizability to scenarios involving other programming languages or execution modalities.

Pass rate and accuracy are the main metrics used to assess model performance. While effective in measuring correctness, they may overlook important qualities such as reasoning efficiency, clarity of intermediate steps, and readability of the reasoning trace. To address this, we provide qualitative analyses of reasoning behaviors to complement quantitative metrics as shown in Figure 1 and 5.

References

Wasi Uddin Ahmad, Sean Narenthiran, Somshubra Majumdar, Aleksander Ficek, Siddhartha Jain, Jocelyn Huang, Vahid Noroozi, and Boris Ginsburg. 2025. Opencodereasoning: Advancing data distillation for competitive coding. *arXiv preprint arXiv:2504.01943*.

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

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

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Claude. 2024. https://www.anthropic.com/news/claude-3-5-sonnet. *Claude*.

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

Kanishk Gandhi, Ayush Chakravarthy, Anikait Singh, Nathan Lile, and Noah D Goodman. 2025. Cognitive behaviors that enable self-improving reasoners, or, four habits of highly effective stars. *arXiv preprint arXiv:2503.01307*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

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

Yi Hu, Xiaojuan Tang, Haotong Yang, and Muhan Zhang. 2024. Case-based or rule-based: how do transformers do the math? *arXiv preprint arXiv:2402.17709*.

HuggingFace. 2025. https://github.com/huggingface/Math-Verify. *HuggingFace*.

inclusionAI. 2025. https://huggingface.co/datasets/inclusionAI/AReaL-boba-Data/blob/main/AReaL-boba-106k.jsonl. *inclusionAI*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Bespoke Labs. 2025. https://huggingface.co/bespokelabs/Bespoke-Stratos-32B. *Bespoke-Stratos*.

Chengpeng Li, Mingfeng Xue, Zhenru Zhang, Jiaxi Yang, Beichen Zhang, Xiang Wang, Bowen Yu, Binyuan Hui, Junyang Lin, and Dayiheng Liu. 2025a. Start: Self-taught reasoner with tools. *arXiv preprint arXiv:2503.04625*.

Xuefeng Li, Haoyang Zou, and Pengfei Liu. 2025b. Torl: Scaling tool-integrated rl. *arXiv preprint arXiv:2503.23383*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

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

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023a. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572.

Siyao Liu. 2024. https://github.com/bytedance/SandboxFusion. *ByteDance*.

Tianqi Liu, Yao Zhao, Rishabh Joshi, Misha Khalman, Mohammad Saleh, Peter J Liu, and Jialu Liu. 2023b. Statistical rejection sampling improves preference optimization. *arXiv preprint arXiv:2309.06657*.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. 2023. Chameleon: Plug-and-play compositional reasoning with large language models. *Advances in Neural Information Processing Systems*, 36:43447–43478.

Zhiyuan Ma, Zhenya Huang, Jiayu Liu, Minmao Wang, Hongke Zhao, and Xin Li. 2025. Automated creation of reusable and diverse toolsets for enhancing llm reasoning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 24821–24830.

MAA. 2025. https://maa.org/math-competitions/american-invitational-mathematics-examination-aime. *American invitational mathematics examination - aime*.

Open-R1. 2025. https://huggingface.co/blog/open-r1/update-3. *Huggingface*.

OpenAI. 2024. https://openai.com/index/hello-gpt-4o/. *OpenAI*.

OpenAI. 2025. https://openai.com/o1/. *OpenAI*.

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. 2023. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. 2025. Toolrl: Reward is all tool learning needs. *arXiv preprint arXiv:2504.13958*.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36:53728–53741.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551.

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

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

Skywork. 2025. https://huggingface.co/datasets/Skywork/Skywork-OR1-RL-Data. *Skywork*.

Qwen Team. 2025. Qwq-32b: Embracing the power of reinforcement learning.

Open Thoughts. 2025. https://huggingface.co/open-thoughts/OpenThinker-32B. *Open Thoughts*.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2024. OpenHands: An Open Platform for AI Software Developers as Generalist Agents.

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

Jeannette M Wing. 2006. Computational thinking. *Communications of the ACM*, 49(3):33–35.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2024. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*.

Hui Yang, Sifu Yue, and Yunzhong He. 2023. Auto-gpt for online decision making: Benchmarks and additional opinions. *arXiv preprint arXiv:2306.02224*.

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

Yixin Ye, Zhen Huang, Yang Xiao, Ethan Chern, Shijie Xia, and Pengfei Liu. 2025. Limo: Less is more for reasoning. *arXiv preprint arXiv:2502.03387*.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, et al. 2025. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*.

Weihao Zeng, Yuzhen Huang, Qian Liu, Wei Liu, Keqing He, Zejun Ma, and Junxian He. 2025. Simplerl-zoo: Investigating and taming zero reinforcement learning for open base models in the wild. *arXiv preprint arXiv:2503.18892*.

Huangzhao Zhang, Kechi Zhang, Zhuo Li, Jia Li, Jia Li, Yongmin Li, Yunfei Zhao, Yuqi Zhu, Fang Liu, Ge Li, et al. 2024a. Deep learning for code generation: a survey. *Science China Information Sciences*, 67(9):191101.

Kechi Zhang, Ge Li, Yihong Dong, Jingjing Xu, Jun Zhang, Jing Su, Yongfei Liu, and Zhi Jin. 2024b. Codedpo: Aligning code models with self generated and verified source code. *arXiv preprint arXiv:2410.05605*.

Kechi Zhang, Ge Li, Jia Li, Yihong Dong, and Zhi Jin. 2025a. Focused-dpo: Enhancing code generation through focused preference optimization on error-prone points. *arXiv preprint arXiv:2502.11475*.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024c. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13643–13658.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023a. Self-edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 769–787.

Kechi Zhang, Huangzhao Zhang, Ge Li, Jia Li, Zhuo Li, and Zhi Jin. 2023b. Toolcoder: Teach code generation models to use api search tools. *arXiv preprint arXiv:2305.04032*.

Kechi Zhang, Huangzhao Zhang, Ge Li, Jinliang You, Jia Li, Yunfei Zhao, and Zhi Jin. 2025b. Sealign: Alignment training for software engineering agent. *arXiv preprint arXiv:2503.18455*.

A Case Study of Computational Thinking Pattern

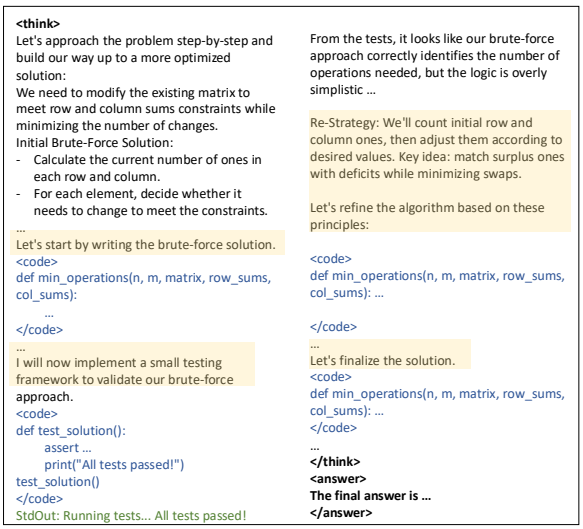
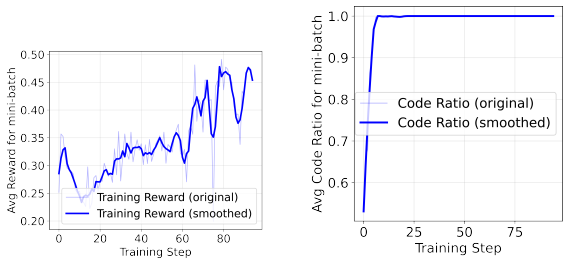


Figure 5: Case Study of the computational thinking pattern for our model. The model shows a *self-imposed validation mechanism* pattern, which enables to detect potential errors and enhance solution robustness.

We collect the generated solutions from our Computational Thinking Model (CTM) and manually analyze them to determine whether they exhibit well-known computational thinking patterns, as illustrated in Figure 5 in Appendix. For this challenging coding problem, our CTM demonstrates a *self-imposed validation mechanism* pattern. The model first generates a brute-force solution along with corresponding test cases, which are validated against this reference implementation. It then iteratively refines the algorithm into a more optimized version while maintaining correctness through this validation framework. This systematic approach enables the model to detect potential errors and enhance solution robustness. It aligns with a core principle of computational thinking (Wing, 2006):



(a) Training Reward (b) Code Reasoning Ratio

Figure 6: Reinforcement learning training curves for our model. The code-reasoning ratio—defined as the proportion of rollouts containing `<code>...</code>` tags—reflects the model's tendency to engage in executable, code-driven reasoning.

> *Computational thinking is reformulating a seemingly difficult problem into one we know how to solve, perhaps by reduction, embedding, transformation, or simulation.*

B Training Curves for Computational Thinking Model

We show the reinforcement learning training curves of our computational thinking model training process, including the training reward curve and the code-reasoning ratio for each mini-batch along the entire training process in Figure 6.

C Interactive Computational Thinking Algorithm

This algorithm outlines the system's interactive reasoning process, where the model alternates between generating reasoning steps, executing code, and integrating execution results as feedback. The sandbox environment is particularly crucial, as it supports persistent memory across code executions, allowing variable states, functions, and intermediate results to persist across iterations. This feature enables the model to build upon its own computations dynamically, reflecting the iterative and modular nature of computational thinking.

D Benchmarks and Metrics

For evaluating model performance on code-related tasks, we use six widely used programming benchmarks. Functional code generation benchmarks contain HumanEval, HumanEval+, MBPP, and MBPP+ (Chen et al., 2021; Austin et al., 2021; Liu et al., 2023a), focusing on functional correctness in code generation with extended test cases. Programming contest benchmarks include CodeContests and LiveCodebench (Li et al., 2022; Jain et al., 2024), where CodeContests has 165 programming code contest questions from diverse platforms, and LiveCodebench has the newest contest questions released between May 2023 and January 2025, containing 880 problems.

For further evaluating reasoning capabilities beyond programming, we apply three benchmarks in mathematical problem-solving. AIME 2024 and AIME 2025 benchmarks (MAA, 2025) contain problems from the American Invitational Mathematics Examination. These problems are prestigious high school mathematics competition problems known for its challenging mathematical problems. MATH-500 (Lightman et al., 2023) is collected from real-world math problems verified by OpenAI, which requires long-horizon reasoning.

To evaluate the performance of CTM, we execute the generated code on the hidden test cases to get the pass rate for each code question. For mathematical tasks, we use the *math-verify* (HuggingFace, 2025) to evaluate the generated output and the ground truth output to get the accuracy for math questions.

E Compared Baselines

We evaluate our approach against three categories of state-of-the-art reasoning systems to provide comprehensive performance comparisons, including proprietary source models, open-source models, and tool-augmented agents.

Proprietary Source Models: We compare CTM against the advanced proprietary source models, containing Anthropic's `Claude-3.5-Sonnet` (Claude, 2024), `OpenAI's GPT-4o-0513` (OpenAI, 2024), and `o1` series models (OpenAI, 2025). They represent the current pinnacle of commercial reasoning capabilities, with some results drawn from official benchmarks where available.

Open-Source Models: These models in our comparison comprise several open-source systems optimized specifically for programming tasks, such as `Bespoke-Stratos-32B` (Labs, 2025), `OpenThinker-32B` (Thoughts, 2025), `OlympicCoder-32B` (Open-R1, 2025), and `OCR-Qwen-32B` (Ahmad et al., 2025), along with `DeepSeek-V3` (Liu et al., 2024) and `R1` (Guo et al., 2025). While these models demonstrate strong performance on coding tasks through inten-

Algorithm 1 Interactive Language and Code Mixed Reasoning in CTM

procedure INTERACTIVECTM(Q: input question)
  $H \leftarrow$ [**initial prompt:** Q] ▷ Start with `<think>`
  while `</think>` not in H **do**
    $o \leftarrow \text{LLM}(H)$ ▷ Model generates next step until get `</code>`
    if `<code>` in o **then**
      `code` $\leftarrow$ extract code block from o
      `result` $\leftarrow$ RUNINSANDBOX(`code`)
      $H \leftarrow H + (o + \texttt{</code>} + \texttt{result})$
    else
      $H \leftarrow H + o$
    end if
  end while
  $ans \leftarrow \text{LLM}(H)$ ▷ Start with `<answer>` until get `</answer>`
  return answer enclosed by `<answer>...</answer>`
end procedure

sive domain-specific training, they typically show limitations when applied to broader mathematical reasoning problems.

Tool-Augmented Agents: We also contrast our model with tool-augmented agent frameworks, where `OpenHands` (Wang et al., 2024) is a representative example. It enhances LLMs' reasoning ability through integrated development environments that combine text editing, code execution, and multi-step planning capabilities. In this paper, we use the advanced `DeepSeek-V3` as the base LLM.

Considering that CTM is continuously trained from `Qwen2.5-32B-Instruct`, we evaluate the performance of `Qwen2.5-32B-Instruct` (Yang et al., 2024) to further analyze the effectiveness of our CTM. All comparisons maintain identical evaluation settings to ensure fair and consistent measurements.

F Implementation Details

For the sandbox, in practice, we choose *Sandbox Fusion* (Liu, 2024), and use its provided *Jupyter* mode for our model, which provides an isolated execution environment. Despite slightly higher latency, it delivers superior stability for sustained training operations. We implemented specific error handling optimizations to enhance training effectiveness. When *Sandbox Fusion* gives out execution errors, it generates verbose tracebacks containing irrelevant file path information. To reduce context length and preserve only relevant error information, we extract only the final line of error messages.

In the training process, we initialize CTM with `Qwen-2.5-32B-Instruct` (Yang et al., 2024), where we fine-tune it under supervised fine-tuning (SFT) on computational thinking-flavored datasets (§3.2) and RL-based fine-tuning for more effective problem-solving behaviors (§3.2). In this paper, all RL-based experiments are conducted using the *veRL* framework, adopting the DAPO algorithm for RL. We sample a batch size of 32 queries per training step, generating 16 responses per query. We utilize a Jupyter-style interactive execution environment. This environment allows the model to generate, execute, and iteratively refine code in real-time. We ultimately selected Sandbox Fusion (Liu, 2024), which provides an isolated execution environment. Despite slightly higher latency, it delivers superior stability for sustained training operations. All experiments are conducted on 32 A100-80GB GPUs. Training typically completes within one week for our experiments under the above configurations.

In the inference process, greedy search is adopted as it ensures deterministic and consistent results for evaluation. Some evaluation results are referenced from their official reports (Yang et al., 2025; Guo et al., 2025; Ahmad et al., 2025).

G Compatibility with Popular RL Algorithms

The principles underlying our computational thinking model are highly versatile and can be adapted to a wide range of advanced reinforcement learn-

ing (RL) techniques. These include offline RL algorithms, such as Reject Sampling (Liu et al., 2023b) and Direct Preference Optimization (DPO) (Rafailov et al., 2023), as well as online RL algorithms, such as Proximal Policy Optimization (PPO) (Schulman et al., 2017), Generalized Policy Optimization (GRPO) (Shao et al., 2024), and their advanced variants (Yu et al., 2025). The key to the generality and applicability of our approach lies in its minimally invasive design. Specifically, our method modifies only the rollout behavior during the model's decoding process, without altering the underlying model architecture or the formulation of the training loss function. This decoupling from the foundational components of model training ensures that our computational thinking framework retains compatibility with a variety of RL paradigms and implementations.

H Further Analysis of Related Work

H.1 Large Reasoning Models

Recent developments in large language models (LLMs) (OpenAI, 2024; Liu et al., 2024) have significantly enhanced their ability to perform reasoning tasks. A cornerstone technique driving this progress is Chain-of-Thought (CoT) prompting, first introduced by (Wei et al., 2022), which decomposes reasoning processes into step-by-step explanations expressed in natural language. CoT prompting has demonstrated remarkable improvements in reasoning tasks, such as mathematical problem-solving, commonsense reasoning, and multi-hop question answering, by enabling models to explicitly articulate intermediate steps. Building on CoT, research has continued to innovate on reasoning mechanisms. Advanced LLMs like OpenAI-o1 (OpenAI, 2025) and DeepSeek-R1 (Guo et al., 2025) demonstrate the effectiveness of CoT-inspired approaches, achieving state-of-the-art performance on benchmarks requiring systematic decomposition and multi-step reasoning. Complementing CoT, the Program-of-Thought (PoT) approach (Chen et al., 2022; Gao et al., 2023) integrates external computational tools—such as Python interpreters—to simplify and validate reasoning operations. The big difference between our method and the PoT method is that our model can reason in an interactive natural language and code mixed reasoning pattern, which can benefit from both language reasoning models and code execution capability.

H.2 Tool-Integrated Models

Building upon the concept of human tool use, recent efforts focus on augmenting LLMs with external computational tools to tackle inherently complex tasks. Tool-integrated reasoning, first introduced to address computationally intensive requirements in domains such as mathematics (Chen et al., 2022; Schick et al., 2023; Yao et al., 2023; Zhang et al., 2023b,a; Li et al., 2025a; Ma et al., 2025; Paranjape et al., 2023; Lu et al., 2023), provides LLMs access to environments for feedback. Early works such as ToolFormer (Schick et al., 2023) showed the potential of incorporating tool execution in reasoning processes. In addition to traditional tool-integrated frameworks, LLMs have played a key role in developing intelligent agents capable of multi-modal task execution. Representative examples include Auto-GPT (Yang et al., 2023), CodeAgent (Zhang et al., 2024c), and OpenHands (Wang et al., 2024; Zhang et al., 2025b), which showcase the potential of hybrid systems leveraging reasoning and tool use in complex task-solving workflows. These advances underscore the importance of combining computational capabilities with reasoning strategies, which serve as the foundation of our work.

In recent concurrent work, reinforcement learning (RL) has emerged as a powerful approach for training tool-integrated reasoning models (Feng et al., 2025; Qian et al., 2025; Li et al., 2025b; Wang et al., 2024). For instance, the ReTool framework (Feng et al., 2025) applies RL to optimize code interpreter usage specifically for mathematical reasoning tasks, and observes relatively simple tool-use patterns, such as basic code self-correction.

Our research advances this line of concurrent work in two key directions. First, we expand the application domain beyond mathematics to encompass more diverse code generation scenarios. Second, and more importantly, we introduce a structured methodology in which computational tools are used not only for executing code, but also as instruments to guide the trajectory of reasoning itself. Unlike these approaches, the Computational Thinking Model (CTM) embeds computational thinking principles directly into its reasoning architecture during the supervised fine-tuning (SFT) phase and further reinforces this strategy through RL training. This design enables greater adaptability, longer-term coherence, and the emergence of higher-level reasoning patterns such as decomposition, recur-

sion, and iterative refinement. Inspired by the foundational concept of computational thinking (Wing, 2006), CTM reframes problem-solving tasks within a paradigm of structural transformation—through simulation, modularization, and algorithmic abstraction—as illustrated in our case studies in Figures 1 and 5. These contributions place CTM at the intersection of tool-augmented reasoning and structured computational thinking paradigms. By combining computational precision with cognitive structure, CTM represents a significant step toward building generalizable, interpretable, and robust intelligent systems capable of solving complex, real-world problems.