

Folding Context into Executable States: Structured Strategy Folding for Long-Horizon Reasoning Agent

Zhengyu Chen^{1*}, Luan Zhang^{1*}, Zhaoyi Li¹, Jinluan Yang²,
Zheqi Lv³, Teng Xiao⁴, Wei Wang¹, Jingang Wang¹

¹Meituan LongCat ²Zhejiang University ³Cornell University ⁴Allen Institute for AI
chencsmat@gmail.com

Abstract

Current Large Language Model (LLM) agents struggle with long-horizon reasoning tasks, such as competition-level mathematics, where sustaining precision over hundreds of steps is critical. A primary bottleneck is the reliance on natural language context or summaries to track progress; these linguistic representations are inherently “lossy,” introducing *semantic drift* and accumulating errors that derail the reasoning chain. To address this, we propose **Structured Strategy Folding (SSF)**, a novel paradigm that shifts the agent’s memory mechanism from linguistic description to computational verification. Instead of compressing history into text, SSF “folds” the reasoning trajectory into a sequence of **Executable Symbolic States (ESS)**. Each ESS serves as a verifiable anchor—comprising either executed code artifacts or formally judged symbolic lemmas—generated through a dual-pathway validation mechanism. By interacting with these rigorous state artifacts rather than raw textual history, the agent effectively eliminates ambiguity and prevents error propagation. Experiments on challenging mathematical benchmarks demonstrate that SSF significantly outperforms standard Chain-of-Thought and existing summarization baselines, achieving substantial accuracy improvements while maintaining robust performance over extended reasoning horizons.

1 Introduction

Recent advancements in Large Language Models (LLMs) (OpenAI et al., 2024; DeepSeek-AI et al., 2025) have demonstrated remarkable capabilities in complex reasoning, largely driven by prompting strategies such as Chain-of-Thought (CoT) (Wei et al., 2022). However, as tasks scale in complexity (Dziri et al., 2023; Shojaei et al., 2025), particularly in domains like competition-level mathematics (Petrov et al., 2025) or software engineering (Jimenez et al., 2024), the reasoning process

extends from a few steps to hundreds. In these long-horizon scenarios, standard agents face a critical limitation: the finite context window and the cognitive load of managing extensive history often lead to “lost-in-the-middle” phenomena (Liu et al., 2024; Laban et al., 2025) and hallucination (Sun et al., 2025; Ye et al., 2025).

To mitigate context overflow, existing approaches (Sun et al., 2025; Ye et al., 2025; Shao et al., 2025) typically employ summarization techniques, compressing past reasoning steps into natural language descriptions. While effective for narrative tasks, we argue that this approach is fundamentally flawed for rigorous symbolic reasoning. Natural language is inherently ambiguous and “lossy”; summarizing a precise mathematical derivation into text often discards crucial state information. We term this phenomenon *semantic drift* (Zanbaghi et al., 2025; Kumar et al., 2025): the gradual divergence of the agent’s internal context from the ground truth due to the accumulation of imprecise linguistic approximations. As the reasoning chain lengthens, this drift compounds, rendering the agent unable to verify previous intermediate results or backtrack effectively (Yeo et al., 2025).

To solve this, we introduce **Structured Strategy Folding (SSF)**, a framework that shifts the agent’s memory paradigm from linguistic summarization to **Executable Symbolic States (ESS)**. Unlike lossy textual summaries, an ESS is a rigorous data structure comprising a *core artifact* (e.g., a code snippet or symbolic lemma) paired with *verification evidence* (e.g., execution output or a logical proof). These states are constructed via a **Dual-Pathway Validator**: computational steps are verified by a deterministic code executor, while abstract logical steps are scrutinized by a semantic judge. By exposing the agent exclusively to these verified anchors rather than the noisy generation history, SSF enforces a “correct-by-construction” memory that remains precise regardless of reasoning depth.

*These authors contributed equally.

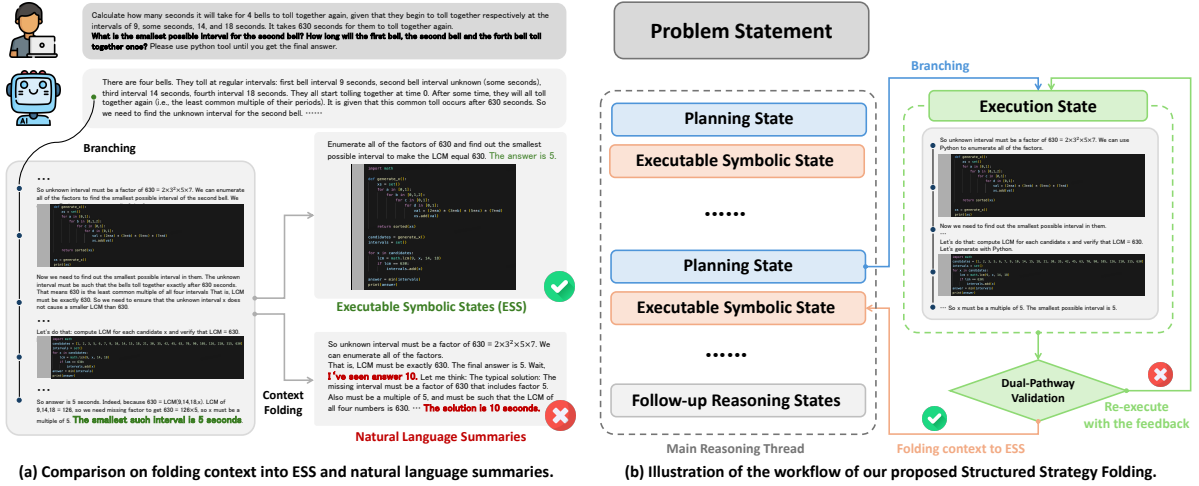


Figure 1: (a): the illustration of folding context into Executable Symbolic States (ESS) and Natural Language Summaries. Folding context into natural language summaries can introduce additional hallucination and inconsistency that do harm to the agents’ reasoning performance. In contrast, folding context into ESS largely guarantee the validity of the compressed content. (b): the illustration of the workflow of the proposed Structured Strategy Folding (SSF). The main reasoning thread contains Planning States and ESS. After the Planning States, the agents branch into executing sub-tasks (Execution State), the generated context of which will be checked by the Dual-Pathway Validation. If the context passes the validation, it will be compressed into ESS and merged to the main reasoning thread. Otherwise, the agent will be asked to re-execute the sub-task with the validation feedback.

SSF empowers the agent to *automatically manage its working context* through a procedural *branch-execute-fold* loop. Rather than relying on fixed summarization schedules or external heuristics, the agent procedurally *branches* into a dedicated sub-trajectory whenever it identifies a sub-task requiring intensive computation or multi-step deduction as shown in Figure 1.

Different from previous context folding methods (Sun et al., 2025; Ye et al., 2025), SSF ensures that *only verifiable results are merged back into the main reasoning thread*. This mechanism effectively filters out intermediate errors and hallucinations, enabling the agent to maintain a minimal, high-fidelity reasoning state even across extended horizons, as shown in Figure 2.

Our contributions are summarized as follows: (1) We identify *semantic drift* in natural language summarization as the primary bottleneck for long-horizon symbolic reasoning agents. (2) We propose **Structured Strategy Folding (SSF)**, a framework that empowers agents to *automatically determine when and how to manage their working context* through procedural branching and folding, replacing textual history with **Executable Symbolic States (ESS)** and enabling loss-less context compression. (3) We introduce a dual-pathway validation mechanism that integrates code execution and semantic judging to ensure that *only verifiable*

results are merged into the main reasoning thread, guaranteeing the reliability of every folded state. (4) Extensive experiments on mathematical benchmarks show that SSF significantly reduces error accumulation, outperforming strong baselines in multi-step reasoning tasks while maintaining robust performance over extended reasoning horizons.

2 Related Work

Tool-Integrated Reasoning for LLM Agents

The paradigm of Tool-Integrated Reasoning (TIR) has emerged as a primary solution (Zhang et al., 2024a; Gou et al., 2024; Du et al., 2025) to overcome the inherent limitations of LLMs in precise arithmetic computation, code simulation, and real-time information retrieval. ReAct (Yao et al., 2023) established the foundational framework by interleaving reasoning traces with task-specific actions, demonstrating that synergy between “thinking” and “acting” reduces hallucinations and improves plan robustness. Building on this, Lin and Xu (2025) provided the first formal proof that tools strictly break the capability ceiling of pure-text models through improved token efficiency. Recent efforts (Li et al., 2025b,a; Xue et al., 2025; Jin et al., 2025) have focused on activating and stabilizing TIR in large reasoning models (LRMs). START (Li et al., 2025b) introduced Hint-infer to stimulate latent tool-use capabilities using artificial hints, while

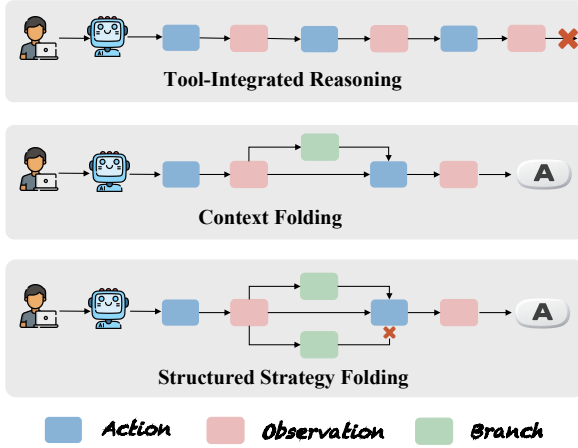


Figure 2: Comparison of our proposed structured strategy folding (SSF) with traditional tool-integrated reasoning (TIR) and context folding methods. In long-horizon reasoning, TIR may fail to obtain the final answer within a limited context window, while context folding alleviates this issue but is prone to *semantic drift*. In contrast, SSF replaces textual summaries with Executable Symbolic States (ESS) and introduces a dual-pathway validation mechanism to ensure the reliability and lossless nature of the folded states.

utilizing Hint-RFT, a self-training framework, to enable models to learn effective tool (e.g., code interpreter) utilization through diverse hit patterns. CoRT (Li et al., 2025a) further refined this by addressing specific inefficiencies such as “delayed computation” and “code result distrust” through a strategic data synthesis framework, strong-to-weak distillation training, and code-integrated RL. To stabilize RL training in multi-turn scenarios, Simple-TIR (Xue et al., 2025) proposed a trajectory filtering mechanism to eliminate “void turns” preventing gradient explosions caused by low-probability tokens inherited from tool feedback. In the domain of search-integrated reasoning, Search-R1 (Jin et al., 2025) extended the RL paradigm to autonomously generate multiple search queries during step-by-step reasoning, while DeepDive (Lu et al., 2025) enhanced long-horizon search through knowledge-graph grounded multi-hop data synthesis.

Context Folding for Long-Horizon Agents As agents tackle longer-horizon multi-turn interaction tasks (Lu et al., 2025; Xue et al., 2025), managing the linear growth of interaction history becomes critical to avoid context saturation and attention dilution (Zhang et al., 2024b). Context-Folding (Sun et al., 2025) pioneered a framework to teach agents to procedurally branch into sub-trajectories for sub-tasks and fold them upon completion, reducing

active context size by 10x while maintaining performance via RL. AgentFold (Ye et al., 2025) expanded this concept by treating context as a dynamic cognitive workspace, learning to execute multi-scale operations: granular and deep condensation for preserving vital details and abstracting multi-step sub-tasks. To address the training stability issue, FoldAct (Shao et al., 2025) introduced separated loss for independent gradient signals on summary and action tokens and context consistency loss to mitigate training collapse. Complementary to these works, Chi et al. (2025) introduced asynchronous thinking (AsyncThink), which employs a fork-join protocol to organize internal thinking into concurrently executable structures and can be viewed as an organizational form of context folding. Together, these works represent a transition from passive history logging to proactive cognitive management in autonomous agents.

While existing context folding relies on lossy natural language summarization to compress history, SSF adopts a “computational folding” paradigm. Different from previous methods (Sun et al., 2025; Ye et al., 2025), SSF ensures that **only verifiable results are merged back into the main reasoning thread**, preventing unvalidated information from corrupting the folded memory. This verification-first approach preserves the reasoning trajectory through ESS, effectively eliminating the semantic drift that typically accumulates during linguistic compression in long-horizon reasoning tasks.

3 Method

Our framework, Structured Strategy Folding (SSF), addresses the fragility of long-horizon reasoning by shifting the memory paradigm from linguistic summarization to **Executable Symbolic States (ESS)**.

3.1 Overview: Automatic Context Management via Branch-Execute-Fold

A central innovation of SSF is that the agent **automatically determines when and how to manage its working context**, without relying on fixed summarization schedules or external heuristics. Rather than passively accumulating all reasoning steps, SSF empowers the agent to procedurally **branch** into a dedicated sub-trajectory whenever it identifies a sub-task requiring intensive computation or multi-step deduction. Within this branch, the agent generates candidate derivations and subjects them to rigorous validation through our Dual-Pathway

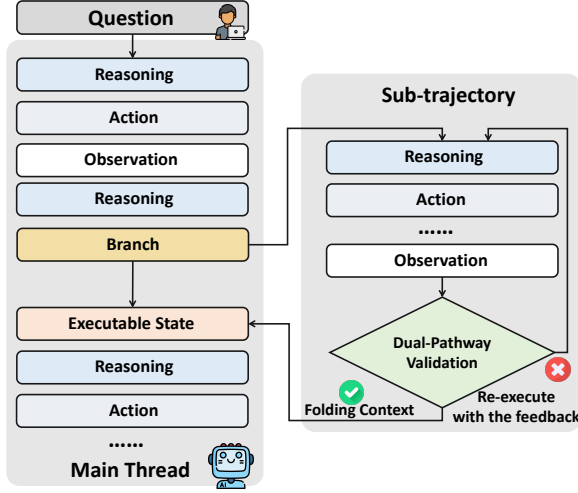


Figure 3: Workflow of Structured Strategy Folding. The agent operates in the main reasoning thread (**Planning State**) and procedurally branches into sub-trajectories (**Execution State**) for intensive sub-tasks. The Dual-Pathway Validator verifies each candidate derivation. Only validated results are folded into ESS and merged back into the main thread.

Validator. *Only verifiable results are merged back into the main reasoning thread*, ensuring that the folded memory contains exclusively validated, immutable anchors. This active context management mechanism, as illustrated in Figure 3, enables the agent to maintain a minimal, verifiable reasoning state while handling long reasoning chains.

The key insight is that by folding verified states rather than linguistic summaries, SSF eliminates the semantic drift that accumulates in traditional context folding methods. Each folded state is a computational or logical artifact with attached verification evidence, making the reasoning trajectory reproducible and auditable. As the agent progresses, the active context remains bounded while the folded memory grows with only verified, immutable anchors—a stark contrast to the linear context growth in standard CoT approaches.

3.2 Problem Formulation: Context Folding

Standard LLM agents model long-horizon trajectories τ as a sequence of actions a_t and observations o_t . The probability of a trajectory given a query q is typically defined as:

$$p_{\theta}(\tau|q) = \prod_{t=1}^T p(o_t|\tau_{<t}, a_t) \pi_{\theta}(a_t|q, \tau_{<t}) \quad (1)$$

In complex reasoning, $\tau_{<t}$ grows linearly, leading to context saturation. We define **Structured Strat-**

egy Folding (SSF) as an agentic framework that replaces the raw history $\tau_{<t}$ with a folded state $\mathcal{F}(\tau_{<t})$. The folding operator $\mathcal{F}$ compresses intermediate reasoning sub-trajectories into **Executable Symbolic States (ESS)**, such that:

$$p_{\theta}^{SSF}(\tau|q) = \prod_{t=1}^T \pi_{\theta}(a_t|q, \mathcal{F}(\tau_{<t})) \quad (2)$$

where $\mathcal{F}(\tau_{<t})$ ensures that the active context contains only the problem statement P and a sequence of verified ESS anchors.

3.3 Architecture Components

The SSF architecture consists of three primary modules designed to manage the transition from abstract planning to concrete execution.

Reasoning Agent ($\mathcal{G}$): We employ an LLM-based agent that operates in two modes. In the **Planning State** (corresponding to the main reasoning thread in Figure 1 and Figure 3), $\mathcal{G}$ performs high-level task decomposition. In the **Execution State** (corresponding to the sub-trajectory branches in the figures), the agent operates within a dedicated sub-branch to generate a *Candidate Derivation* $\hat{d}_t$.

Dual-Pathway Validator ($\mathcal{V}$): To ensure the “correct-by-construction” property of the memory, every candidate d_t must pass a validator before being folded: **Deterministic Executor ($\mathcal{V}_{exec}$):** Handles code-based derivations (e.g., Python). It verifies syntax and runtime success, returning the computed result r_t^{exec} . **Semantic Judge ($\mathcal{V}_{judge}$):** Evaluates logical assertions or symbolic lemmas that cannot be executed. It acts as a formal critic to verify that $\hat{d}_t$ follows logically from the previous states $\mathcal{M}_{t-1}$.

Constructing Executable Symbolic States. An ESS is formed only when a candidate $\hat{d}_t$ passes the validator. Once validated, $\hat{d}_t$ is committed as the verified derivation $d_t \leftarrow \hat{d}_t$. Formally, an ESS at step t is a tuple $s_t = (d_t, r_t)$, where d_t is the verified derivation (code/plan) and r_t is the validation evidence (execution output or the judge’s confirmation, i.e., $r_t = r_t^{exec}$ or $r_t = r_t^{judge}$).

3.4 Iterative Reasoning Protocol: The Branch-Execute-Fold Protocol

The inference process of SSF proceeds as a sequential state transition. Let $\mathcal{M}_{t-1} = [s_1, s_2, \dots, s_{t-1}]$ denote the sequence of folded ESSs accumulated up to step $t - 1$. At step t , the system executes the following protocol:

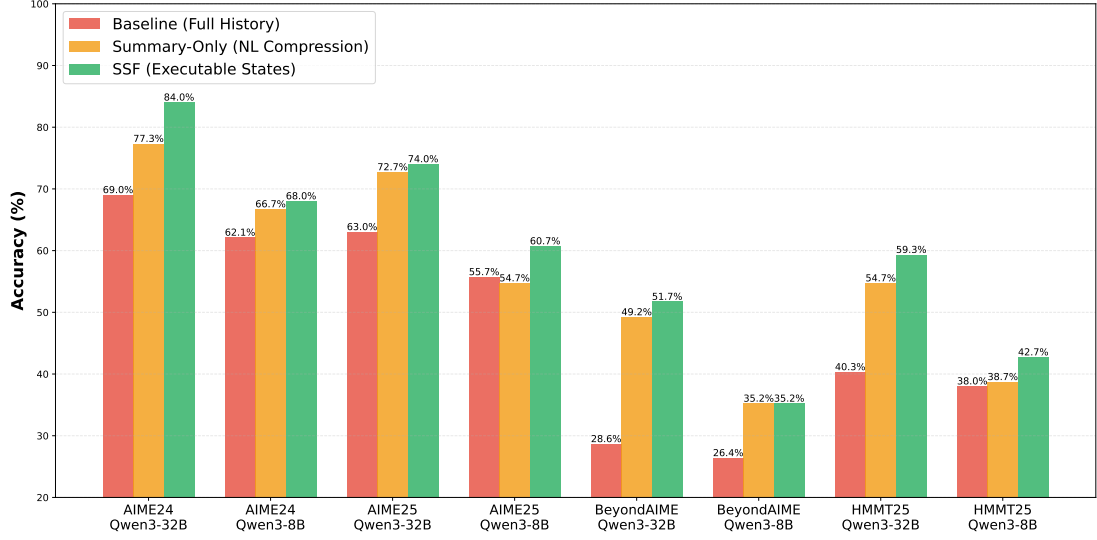


Figure 4: Accuracy comparison across eight experimental configurations: full history baseline, linguistic summarization, and executable symbolic states.

Procedural Branching: When the agent identifies a sub-task requiring intensive computation or multi-step deduction, it invokes the `branch(description, prompt)` tool. This creates a temporary workspace. The description serves as a semantic tag for the sub-task, while the prompt provides the necessary constraints to the agent’s execution mode.

The reasoning agent $\mathcal{G}$ observes the problem statement P and the folded memory $\mathcal{M}_{t-1}$. It generates a candidate derivation $\hat{d}_t$ aimed at advancing the solution:

$$\hat{d}_t \leftarrow \mathcal{G}(P, \mathcal{M}_{t-1}) \quad (3)$$

Here, $\hat{d}_t$ is structured either as a code block (for calculation) or a text plan (for logical deduction).

Execution and Validation: Within the branch, the agent generates candidate derivations. Unlike standard CoT, these steps are local to the branch. The candidate $\hat{d}_t$ is routed to the Dual-Pathway Validation module in $\mathcal{V}$:

- **Case A: Code Derivation.** If $\hat{d}_t$ involves computation, $\mathcal{V}_{exec}$ executes the code. If execution succeeds (exit code 0) and yields a result r_t^{exec} , validation passes ($r_t = r_t^{exec}$). If it fails (e.g., `SyntaxError` or `Timeout`), the error message is returned to $\mathcal{G}$ for refinement.
- **Case B: Strategic Plan/Symbolic Logic.** If $\hat{d}_t$ is a logical assertion, $\mathcal{V}_{judge}$ evaluates its soundness by checking whether it follows logically from $\mathcal{M}_{t-1}$. If the judge returns *APPROVED*,

validation passes ($r_t = r_t^{judge}$). If the judge returns *REJECTED* with a feedback explanation, the feedback is returned to $\mathcal{G}$ to trigger a self-correction loop.

State Folding (The Commit Step): Upon successful validation, the entire sub-trajectory of the branch is discarded from the active context, and the validated candidate $d_t \leftarrow \hat{d}_t$ together with its validation evidence r_t (i.e., r_t^{exec} or r_t^{judge}) are “folded” into a new Executable Symbolic State $s_t = (d_t, r_t)$. This state is then appended to the memory: $\mathcal{M}_t \leftarrow \mathcal{M}_{t-1} \oplus s_t$. Crucially, if validation fails, the system triggers a Self-Correction Loop, prompting $\mathcal{G}$ to regenerate d_t based on the feedback, preventing error propagation.

3.5 State Rendering and Context Folding

The efficiency of SSF stems from how the folding operator $\mathcal{F}$ renders the history. In standard Chain-of-Thought (CoT), the context grows linearly with verbose natural language tokens. In SSF, we define a **State Rendering Function** $\mathcal{R}(s_i)$ that transforms a verbose execution history into a compact artifact:

$$\mathcal{F}(\tau_{<t}) = P \oplus [\mathcal{R}(s_1), \mathcal{R}(s_2), \dots, \mathcal{R}(s_{t-1})] \quad (4)$$

- **Computational ESS:** $\mathcal{R}$ displays the high-level intent and the resulting variable bindings (e.g., `ans = 42`), omitting the raw code and `stderr` logs.
- **Logical ESS:** $\mathcal{R}$ displays the verified lemma or the specific milestone reached.

By rolling back the KV-cache to the point of the branch call and appending only the rendered

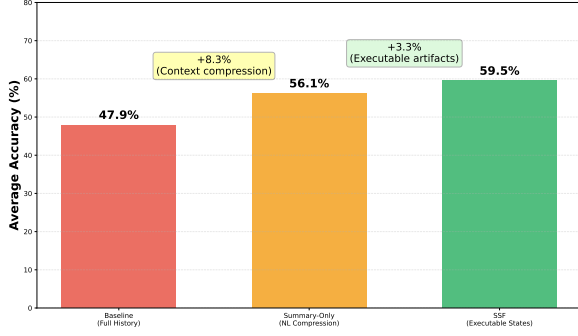


Figure 5: Improvement breakdown showing the contribution of context compression (8.3%) versus executable state representation (11.6%) to the total improvement over baseline.

$\mathcal{R}(s_i)$, SSF maintains a minimal active context window. This prevents *semantic drift*—the accumulation of linguistic errors—by ensuring that every piece of information in the agent’s memory has been computationally or semantically verified.

4 Experiments

4.1 Experimental Setup

We compare our methods with the competitive baselines, including ZeroTIR (Mai et al., 2025), SimpleTIR (Xue et al., 2025), Retool (Feng et al., 2025), ARPO (Dong et al., 2025), Effective TIR (Bai et al., 2025), ToRL (Li et al., 2025c), OpenAI o1-preview (OpenAI, 2024), and QWQ-32B (Team Qwen, 2025). We evaluate SSF on four challenging mathematical reasoning benchmarks (AIME24, AIME25, BeyondAIME, and HMMT25) using two model scales (Qwen3-8B and Qwen3-32B). Detailed dataset and model information are provided in Appendix A. We compare SSF against two methods: **Baseline (Full History)** which preserves all reasoning steps without compression, and **Summary-Only (Linguistic Compression)** which compresses history into natural language summaries. All experiments use a maximum of 50 turns per problem with code execution performed via a sandboxed Python interpreter.

4.2 Experimental Results

To provide comprehensive empirical validation of SSF’s core claims, we present a series of detailed experiments that quantify the method’s advantages across multiple dimensions.

Linguistic Summarization vs. Executable States. A central claim of SSF is that natural language summarization is inherently lossy, while executable

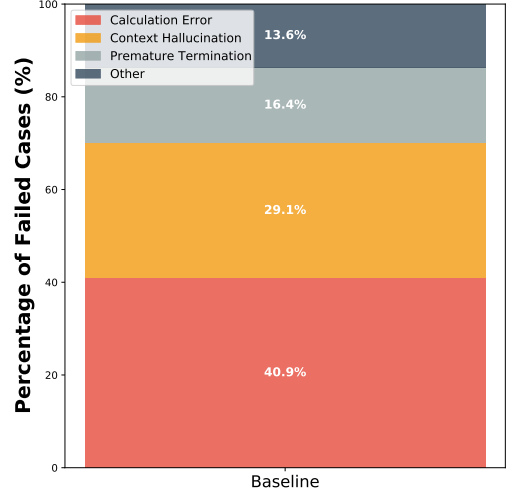


Figure 6: Error distribution across failure cases. The baseline exhibits high context hallucination (29.1% of failures), while SSF reduces this to 11.8%, an 60% relative reduction. This directly quantifies the semantic drift phenomenon and validates SSF’s core mechanism.

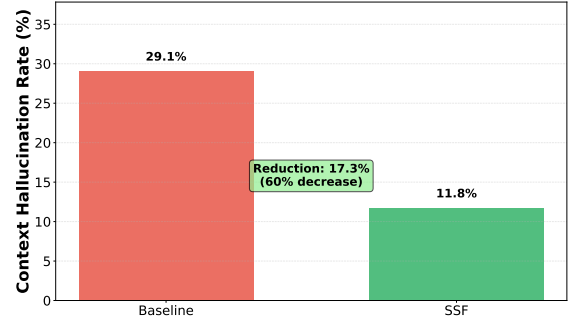


Figure 7: Context hallucination error rates: baseline vs. SSF. The dramatic reduction from 29.1% to 11.8% provides direct empirical evidence that Executable Symbolic States prevent semantic drift and maintain constraint preservation.

symbolic states provide lossless compression. Figure 4 compares three approaches across eight experimental configurations. Baseline (Full History): preserves all reasoning steps without compression (47.89% average accuracy). Summary-Only (Linguistic Compression): compresses history into natural language summaries (56.15% average accuracy, +8.26% improvement). SSF (Executable States): Folds history into Executable Symbolic States (59.45% average accuracy, +3.30% additional improvement). The improvement breakdown in Figure 5 reveals that a substantial portion of SSF’s total improvement comes from executable state representation rather than simple context compression. This directly validates that the *verifiability* and *structure* of ESS—not merely compression—drives the performance gains. The remain-

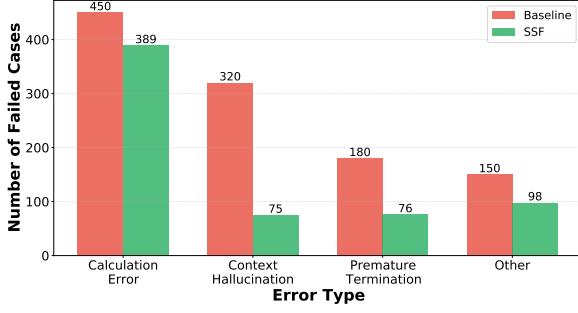


Figure 8: Absolute error counts by category.

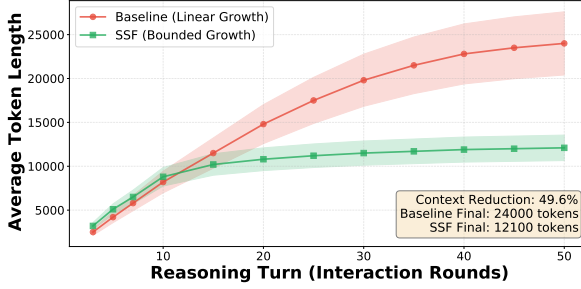


Figure 9: Context window growth over reasoning turns.

ing improvement comes from the implicit context reduction, demonstrating the synergy between compression and verification.

Quantifying Semantic Drift: Error Analysis. To directly measure the semantic drift phenomenon, we analyze error distributions across failure cases. Figure 6 breaks down errors into four categories: *Calculation Errors*: arithmetic or computational mistakes. *Context Hallucination*: forgetting or misrepresenting problem constraints due to context loss. *Premature Termination*: stopping before reaching a valid solution. *Other*: other errors.

The most striking finding is the dramatic reduction in context hallucination errors. In the baseline, context hallucination accounts for 29.1% of failures (320 out of 1,100 cases). In SSF, this drops to just 11.8% of failures (75 out of 638 cases)—an 59.5% relative reduction. This quantitative evidence directly validates our core hypothesis: by replacing linguistic summarization with Executable Symbolic States, SSF prevents the accumulation of semantic drift and maintains verifiable problem constraints throughout extended reasoning chains.

Figure 7 provides a direct comparison of context hallucination rates, while Figure 8 shows the absolute error counts, emphasizing the scale of the improvement. Table 1 presents the accuracy of SSF and the baseline across all benchmarks.

Overall Effectiveness of SSF. Across all configurations, SSF substantially outperform the

Method	AIME24	AIME25	HMMT25
<i>TIR Baselines</i>			
OpenAI o1-preview	44.6	37.9	-
QWQ-32B-Preview	50.0	33.5	-
ARPO-7B	30.0	30.0	-
Effective TIR-7B	42.3	29.2	-
ToRL-7B	40.2	27.9	-
ReTool-32B	67.0	49.3	-
ZeroTIR-32B	48	27	20
SimpleTIR-32B	59.9	49.2	34.6
<i>Qwen3-32B</i>			
Qwen3-32B-TIR	69.0	63.0	40.3
Summary-Only	77.3	72.7	54.7
SSF	84.0	74.0	59.3
<i>Qwen3-8B</i>			
Qwen3-8B-TIR	62.1	55.7	38.0
Summary-Only	66.7	54.7	38.7
SSF	68.0	60.7	42.7

Table 1: Accuracy (%) on mathematical reasoning benchmarks. SSF consistently outperform the baseline on different benchmarks and model scales.

Configuration	AIME24 Accuracy
SSF	84.0
w/o Summary (code only)	74.3
w/o Code (summary only)	77.3
w/o $\mathcal{V}_{exec}$ (judge only)	80.1

Table 2: Ablation study on AIME24 (Qwen3-32B). Removing any core component significantly degrades performance, confirming the necessity of both symbolic execution and semantic summarization.

baseline. On Qwen3-32B, SSF achieves an gain of **+15.0%** on AIME24 and **+11.0%** on AIME25 compared to the baseline. This validates our core hypothesis that replacing verbose textual history with Executable Symbolic States (ESS) effectively mitigates semantic drift in long-horizon reasoning.

Ablation Study We conduct ablations on AIME24 using Qwen3-32B (see Table 2): **w/o Summary**: Accuracy drops to 74.3%. This confirms that natural language summaries are essential for maintaining high-level strategic coherence across long reasoning chains. **w/o Code**: Retaining only summaries yields 77.3%, demonstrating that executable artifacts are crucial for grounding the reasoning in verifiable computational states. **w/o $\mathcal{V}_{exec}$** : Disabling code execution and relying solely on $\mathcal{V}_{judge}$ reduces accuracy to 80.1%. This highlights that the dual-pathway validator is not redundant—deterministic verification for computational steps provides a reliability guarantee that

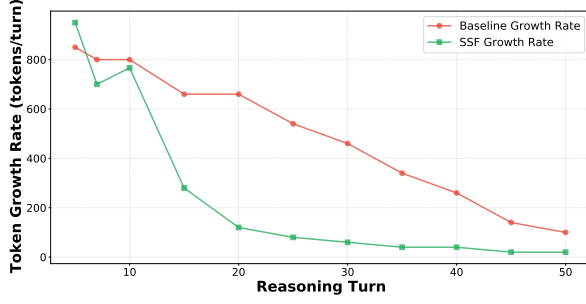


Figure 10: Token growth rate comparison.

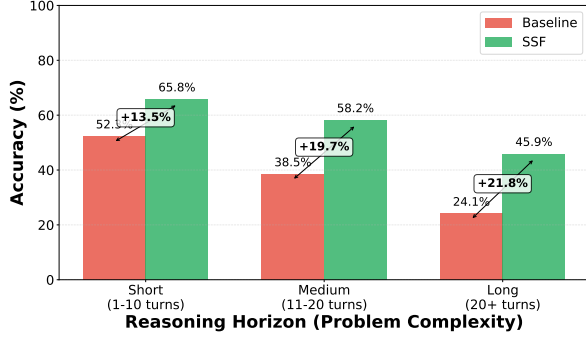


Figure 11: Accuracy stratified by reasoning horizon. SSF’s advantage grows with problem complexity.

semantic judging alone cannot achieve. These validate our architectural design: SSF requires the synergy of *executable validation* (for precision) and *semantic compression* (for coherence).

Context Efficiency and Bounded Growth. Figure 9 illustrates the fundamental efficiency gain of SSF. The baseline approach exhibits linear context growth ($O(n)$) as reasoning steps accumulate, growing from 2,500 tokens at 3 turns to 24,000 tokens at 50 turns (9.6x increase). In contrast, SSF demonstrates bounded growth ($O(1)$), increasing from 3,200 tokens to only 12,100 tokens over the same horizon (3.8x increase). This 49.6% reduction in context window size directly validates our claim that the folding operator $\mathcal{F}$ maintains a minimal active context by replacing verbose reasoning history with compact Executable Symbolic States. Figure 10 further illustrates the diverging growth rates: the baseline’s token growth rate remains consistently high, while SSF’s growth rate decelerates significantly after the initial reasoning phase, reflecting the effectiveness of state folding.

Performance Across Reasoning Horizons. SSF is specifically designed to address long-horizon reasoning challenges. Figure 11 stratifies performance by reasoning complexity: *Short (1-10 turns)*: Baseline 52.3% $\rightarrow$ SSF 65.8% (+13.5% improvement).

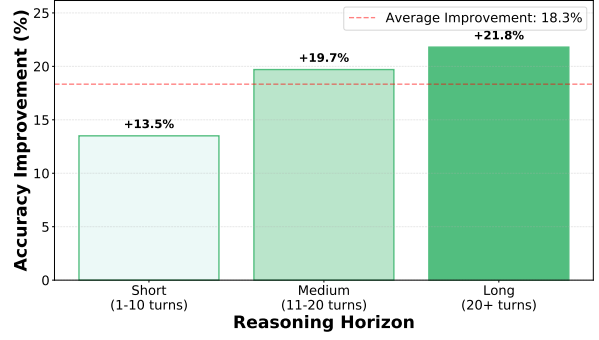


Figure 12: Improvement margin over baseline across reasoning horizons.

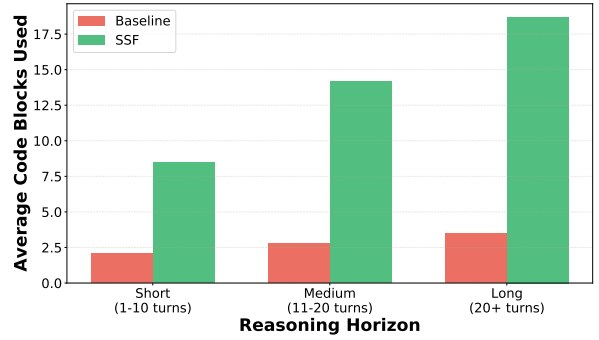


Figure 13: Code block usage by reasoning horizon.

Medium (11-20 turns): Baseline 38.5% $\rightarrow$ SSF 58.2% (+19.7% improvement). *Long (20+ turns)*: Baseline 24.1% $\rightarrow$ SSF 45.9% (+21.8%).

The improvement trend demonstrates that SSF’s advantage grows with reasoning complexity. The largest gains (+21.8%) occur in the long-horizon regime, confirming that SSF specifically alleviates the semantic drift problem that becomes critical in extended reasoning chains. Figure 12 visualizes this trend, while Figure 13 shows that SSF agents employ code verification more consistently across all reasoning horizons, with 5x more code blocks in long-horizon scenarios compared to the baseline.

5 Conclusion

We introduce **Structured Strategy Folding (SSF)**, a novel paradigm for managing context in long-horizon reasoning agents. The core innovation is a fundamental shift from *linguistic summarization* to *computational folding*: instead of compressing history into lossy natural language descriptions, SSF folds the reasoning trajectory into a **Executable Symbolic States (ESS)**, where each state is a verified, immutable artifact containing both the derivation and its validation evidence. Experimental results demonstrate that SSF outperforms baselines.

6 Limitations

While SSF demonstrates significant improvements, we acknowledge several limitations:

1. Dependency on Executable Environments:

SSF relies on access to a deterministic code executor for computational validation. This limits its applicability to domains where code execution is feasible (e.g., mathematical reasoning, algorithm design). For tasks that are inherently non-computational (e.g., open-ended creative writing, subjective judgment), the semantic judge alone may be insufficient, and SSF’s benefits diminish.

2. Semantic Judge Reliability:

The Semantic Judge (V_{judge}) is implemented as the same base model in self-critique mode. This approach may inherit the model’s biases and reasoning limitations. For complex logical reasoning or domain-specific knowledge, the judge may fail to detect subtle errors or inconsistencies. Future work should explore more robust semantic validation mechanisms (e.g., formal verification, external knowledge bases).

3. Agent Autonomy in Branching Decisions:

Currently, the agent’s decisions about when and how to branch are guided by prompts and learned patterns. There is no explicit mechanism to ensure that the branching strategy is optimal. The agent might branch too frequently (increasing overhead) or too infrequently (missing opportunities for verification). A learning-based approach to optimize branching decisions would be valuable.

4. Scalability to Very Long Horizons:

While SSF achieves bounded context growth, the folded memory $\mathcal{M}_t$ still grows linearly with the number of steps. For extremely long reasoning chains (e.g., 1000+ steps), even the folded memory could become unwieldy. Techniques for hierarchical folding or memory summarization at higher levels could address this.

7 Ethics Statement

SSF enhances the reasoning capabilities of LLM agents, enabling them to solve complex, long-horizon problems more reliably. This has several positive implications:

- **Improved Transparency:** By maintaining verifiable, executable artifacts, SSF makes the agent’s reasoning more transparent and auditable. Users and stakeholders can inspect

and verify the reasoning chain, promoting trust and accountability.

- **Reduced Hallucination:** SSF significantly reduces hallucination and semantic drift by grounding reasoning in verified states. This leads to more reliable outputs, which is critical for high-stakes applications (e.g., medical reasoning, legal analysis, scientific discovery).
- **Accessibility and Efficiency:** By enabling smaller models to achieve competitive performance, SSF makes advanced reasoning capabilities more accessible to resource-constrained settings (e.g., edge devices, developing regions). This democratizes access to reasoning AI.
- **Safer Automation:** In automated decision-making systems, SSF’s verification mechanism ensures that decisions are grounded in verified reasoning, reducing the risk of erroneous or biased outcomes.

This has several potential risks:

- **Misuse in Automated Decision-Making:** SSF could be misused to automate high-stakes decisions (e.g., loan approvals, criminal sentencing) without adequate human oversight. While SSF improves reasoning reliability, it does not eliminate the need for human judgment in consequential decisions. We recommend that SSF-based systems be deployed only with explicit human-in-the-loop mechanisms and rigorous impact assessments.
- **Dual-Use Concerns:** SSF could be used to automate harmful activities (e.g., generating adversarial examples, planning attacks, creating misleading content). The verification mechanism, while improving reliability, does not prevent misuse for harmful purposes. We recommend access controls, usage monitoring, and collaboration with policy makers to establish appropriate guardrails.

We recommend the following practices for responsible deployment of SSF:

1. **Transparency:** Clearly communicate to users that SSF-based systems use automated reasoning and verification. Disclose the limitations and potential failure modes.

2. **Human Oversight:** Maintain human-in-the-loop mechanisms, especially for high-stakes applications. Humans should review and approve critical decisions.
3. **Bias and Fairness Audits:** Regularly audit SSF-based systems for bias, fairness, and alignment with values. Use diverse evaluation datasets and stakeholder feedback.
4. **Documentation and Explainability:** Provide detailed documentation of the reasoning process, including the folded states and verification evidence. Enable users to understand and inspect the reasoning chain.
5. **Continuous Monitoring:** Monitor deployed systems for unexpected behaviors, failures, or misuse. Establish feedback loops to improve the system over time.
6. **Collaboration with Stakeholders:** Engage with domain experts, ethicists, policy makers, and affected communities to understand context-specific risks and develop appropriate safeguards.

References

- Fei Bai, Yingqian Min, Beichen Zhang, Zhipeng Chen, Wayne Xin Zhao, Lei Fang, Zheng Liu, Zhongyuan Wang, and Ji-Rong Wen. 2025. Towards effective code-integrated reasoning. *arXiv preprint arXiv:2505.24480*.
- Zewen Chi, Li Dong, Qingxiu Dong, Yaru Hao, Xun Wu, Shaohan Huang, and Furu Wei. 2025. [The era of agentic organization: Learning to organize with language models](#). *Preprint*, arXiv:2510.26658.
- DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, and 181 others. 2025. [Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning](#). *Preprint*, arXiv:2501.12948.
- Guanting Dong, Hangyu Mao, Kai Ma, Licheng Bao, Yifei Chen, Zhongyuan Wang, Zhongxia Chen, Jiazhen Du, Huiyang Wang, Fuzheng Zhang, and 1 others. 2025. Agentic reinforced policy optimization. *arXiv preprint arXiv:2507.19849*.
- Wei Du, Shubham Toshniwal, Branislav Kisacanian, Sadegh Mahdavi, Ivan Moshkov, George Armstrong, Stephen Ge, Edgar Minasyan, Feng Chen, and Igor Gitman. 2025. [Nemotron-math: Efficient long-context distillation of mathematical reasoning from multi-mode supervision](#). *Preprint*, arXiv:2512.15489.
- Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena D. Hwang, Soumya Sanyal, Xiang Ren, Allyson Ettinger, Zaid Harchaoui, and Yejin Choi. 2023. [Faith and fate: Limits of transformers on compositionality](#). In *Thirty-seventh Conference on Neural Information Processing Systems*.
- Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjuan Zhong. 2025. Retool: Reinforcement learning for strategic tool use in llms. *arXiv preprint arXiv:2504.11536*.
- Zhibin Gou, Zhihong Shao, Yeyun Gong, yelong shen, Yujia Yang, Minlie Huang, Nan Duan, and Weizhu Chen. 2024. [ToRA: A tool-integrated reasoning agent for mathematical problem solving](#). In *The Twelfth International Conference on Learning Representations*.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. [SWE-bench: Can language models resolve real-world github issues?](#) In *The Twelfth International Conference on Learning Representations*.
- Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Serkan O Arik, Dong Wang, Hamed Zamani, and Jiawei Han. 2025. [Search-r1: Training LLMs to reason and leverage search engines with reinforcement learning](#). In *Second Conference on Language Modeling*.
- Vivek Kumar, Pushpraj Singh Rajawat, and Eirini Ntoutsi. 2025. [Mitigating semantic drift: Evaluating llms’ efficacy in psychotherapy through mi dialogue summarization](#). *Preprint*, arXiv:2511.22818.
- Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. 2025. [LLMs get lost in multi-turn conversation](#). *Preprint*, arXiv:2505.06120.
- Chengpeng Li, Zhengyang Tang, Ziniu Li, Mingfeng Xue, Keqin Bao, Tian Ding, Ruoyu Sun, Benyou Wang, Xiang Wang, Junyang Lin, and Dayiheng Liu. 2025a. [Teaching language models to reason with tools](#). In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.
- Chengpeng Li, Mingfeng Xue, Zhenru Zhang, Jiaxi Yang, Beichen Zhang, Bowen Yu, Binyuan Hui, Junyang Lin, Xiang Wang, and Dayiheng Liu. 2025b. [START: Self-taught reasoner with tools](#). In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 13523–13564, Suzhou, China. Association for Computational Linguistics.

- Xuefeng Li, Haoyang Zou, and Pengfei Liu. 2025c. Torl: Scaling tool-integrated rl. *arXiv preprint arXiv:2503.23383*.
- Heng Lin and Zhongwen Xu. 2025. [Understanding tool-integrated reasoning](#). *Preprint*, arXiv:2508.19201.
- Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. [Lost in the middle: How language models use long contexts](#). *Transactions of the Association for Computational Linguistics*, 12:157–173.
- Rui Lu, Zhenyu Hou, Zihan Wang, Hanchen Zhang, Xiao Liu, Yujiang Li, Shi Feng, Jie Tang, and Yuxiao Dong. 2025. [Deepdive: Advancing deep search agents with knowledge graphs and multi-turn rl](#). *Preprint*, arXiv:2509.10446.
- Xinji Mai, Haotian Xu, Zhong-Zhi Li, Xing W, Weinong Wang, Jian Hu, Yingying Zhang, and Wenqiang Zhang. 2025. [Agent rl scaling law: Agent rl with spontaneous code execution for mathematical problem solving](#). *Preprint*, arXiv:2505.07773.
- OpenAI, :, Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, Alex Iftimie, Alex Karpenko, Alex Tachard Passos, Alexander Neitz, Alexander Prokofiev, Alexander Wei, Allison Tam, and 244 others. 2024. [Openai o1 system card](#). *Preprint*, arXiv:2412.16720.
- OpenAI. 2024. [Learning to Reason with LLMs](#).
- Ivo Petrov, Jasper Dekoninck, Lyuben Baltadzhiev, Maria Drencheva, Kristian Minchev, Mislav Balunovic, Nikola Jovanović, and Martin Vechev. 2025. [Proof or bluff? evaluating LLMs on 2025 USA math olympiad](#). In *2nd AI for Math Workshop @ ICML 2025*.
- Jiaqi Shao, Yufeng Miao, Wei Zhang, and Bing Luo. 2025. [Foldact: Efficient and stable context folding for long-horizon search agents](#). *Preprint*, arXiv:2512.22733.
- Parshin Shojaee, Seyed Iman Mirzadeh, Keivan Alizadeh, Maxwell Horton, Samy Bengio, and Mehrdad Farajtabar. 2025. [The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity](#). In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.
- Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. 2025. [Scaling long-horizon llm agent via context-folding](#). *Preprint*, arXiv:2510.11967.
- Team Qwen. 2025. [QwQ-32B: Embracing the Power of Reinforcement Learning](#). Blog post.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. [Chain-of-thought prompting elicits reasoning in large language models](#). In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc.
- Zhenghai Xue, Longtao Zheng, Qian Liu, Yingru Li, Xiaosen Zheng, Zejun Ma, and Bo An. 2025. [Simpletir: End-to-end reinforcement learning for multi-turn tool-integrated reasoning](#). *Preprint*, arXiv:2509.02479.
- 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)*.
- Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Zile Qiao, Xinyu Wang, Pengjun Xie, Fei Huang, Siheng Chen, Jingren Zhou, and Yong Jiang. 2025. [Agentfold: Long-horizon web agents with proactive context management](#). *Preprint*, arXiv:2510.24699.
- Edward Yeo, Yuxuan Tong, Morry Niu, Graham Neubig, and Xiang Yue. 2025. [Demystifying long chain-of-thought reasoning in llms](#). *Preprint*, arXiv:2502.03373.
- Shahin Zanbaghi, Ryan Rostampour, Farhan Abid, and Salim Al Jarmakani. 2025. [Detecting sleeper agents in large language models via semantic drift analysis](#). *Preprint*, arXiv:2511.15992.
- Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024a. [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, Bangkok, Thailand. Association for Computational Linguistics.
- Xuechen Zhang, Xiangyu Chang, Mingchen Li, Amit Roy-Chowdhury, Jiasi Chen, and Samet Oymak. 2024b. [Selective attention: Enhancing transformer through principled context control](#). In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

A Experimental Details

A.1 Datasets

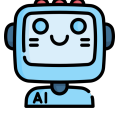
We evaluate SSF on four challenging mathematical reasoning benchmarks that require extensive multi-step derivations:

- **AIME 2024 (AIME24):** The American Invitational Mathematics Examination from 2024, containing 30 competition-level problems¹.

¹<https://aime24.aimeclinic.info/>



Calculate how many seconds it will take for 4 bells to toll together again, given that they begin to toll together respectively at the intervals of 9, some seconds, 14, and 18 seconds. It takes 630 seconds for them to toll together again. **What is the smallest possible interval for the second bell? How long will the first bell, the second bell and the forth bell toll together once?** Please use python tool until you get the final answer.



There are four bells. They toll at regular intervals: first bell interval 9 seconds, second bell interval unknown (some seconds), third interval 14 seconds, fourth interval 18 seconds. They all start tolling together at time 0. After some time, they will all toll together again (i.e., the least common multiple of their periods). It is given that this common toll occurs after 630 seconds. So we need to find the unknown interval for the second bell.

Branching

...
So unknown interval must be a factor of $630 = 2 \times 3^2 \times 5 \times 7$. We can enumerate all of the factors to find the smallest possible interval of the second bell. We

```
def generate_x():
    xs = set()
    for a in (9,11):
        for b in (9,1,2):
            for c in (9,11):
                for d in (9,11):
                    val = (2*9a) + (3*9b) + (5*9c) + (7*9d)
                    xs.add(val)
    return sorted(xs)

xs = generate_x()
print(xs)
```

Now we need to find out the smallest possible interval in them. The unknown interval must be such that the bells toll together exactly after 630 seconds. That means 630 is the least common multiple of all four intervals. That is, LCM must be exactly 630. So we need to ensure that the unknown interval x does not cause a smaller LCM than 630.

Let's do that: compute LCM for each candidate x and verify that $\text{LCM} = 630$.

```
import math
candidates = [1, 2, 3, 5, 6, 7, 9, 10, 14, 15, 18, 21, 26, 35, 42, 45, 63, 70, 90, 105, 126, 210, 315, 630]
intervals = set()
for x in candidates:
    lca = math.lcm(9, x, 14, 18)
    if lca == 630:
        intervals.add(x)
answer = min(intervals)
print(answer)
```

...
So answer is 5 seconds. Indeed, because $630 = \text{LCM}(9, 14, 18, x)$. LCM of 9, 14, 18 = 126, so we need missing factor to get $630 = 126 \times 5$, so x must be a multiple of 5. **The smallest such interval is 5 seconds.**

Enumerate all of the factors of 630 and find out the smallest possible interval to make the LCM equal 630. **The answer is 5.**

```
import math
def generate_x():
    xs = set()
    for a in (9,11):
        for b in (9,1,2):
            for c in (9,11):
                for d in (9,11):
                    val = (2*9a) + (3*9b) + (5*9c) + (7*9d)
                    xs.add(val)
    return sorted(xs)

candidates = generate_x()
intervals = set()
for x in candidates:
    lca = math.lcm(9, x, 14, 18)
    if lca == 630:
        intervals.add(x)
answer = min(intervals)
print(answer)
```

(a) Folding context into **Executable Symbolic States (ESS)**

So unknown interval must be a factor of $630 = 2 \times 3^2 \times 5 \times 7$. We can enumerate all of the factors.

That is, LCM must be exactly 630. The final answer is 5. Wait, **I've seen answer 10**. Let me think: The typical solution: The missing interval must be a factor of 630 that includes factor 5. Also must be a multiple of 5, and must be such that the LCM of all four numbers is 630. ... **The solution is 10 seconds.**

(b) Folding context into **lossy natural language summaries**

Context Folding

follow-up response

Figure 14: The illustration of folding context into Executable Symbolic States (ESS) and Natural Language Summaries. Folding context into natural language summaries can introduce additional hallucination and inconsistency that do harm to the agents' reasoning performance. In contrast, folding context into ESS largely guarantee the validity of the compressed content.

- **AIME 2025 (AIME25):** The 2025 edition of AIME with 30 problems of similar difficulty².
- **BeyondAIME:** An extended benchmark comprising 100 problems that exceed AIME difficulty, designed to stress-test long-horizon reasoning capabilities³.
- **HMMT 2025 (HMMT25):** A benchmark containing 30 problems from HMMT February 2025 used for the MathArena Leaderboard.

A.2 Models

We conduct experiments using two model scales:

²<https://aime25.aimeeducation.info/>

³<https://huggingface.co/datasets/ByteDance-Seed/BeyondAIME>

- **Qwen3-32B:** A 32-billion parameter reasoning agent fine-tuned for mathematical problem-solving⁴.
- **Qwen3-8B:** An 8-billion parameter variant used to assess the scalability of SSF to smaller models⁵.

We compare our methods with the competitive baselines, including Retool (Feng et al., 2025), ARPO (Dong et al., 2025), Effective TIR (Bai et al., 2025), ToRL (Li et al., 2025c), OpenAI o1-preview (OpenAI, 2024), and QWQ-32B (Team Qwen, 2025).

⁴<https://huggingface.co/Qwen/Qwen3-32B>

⁵<https://huggingface.co/Qwen/Qwen3-8B>

A.3 Methods

We compare the following methods:

- **FoldAgent (Summary-Only):** A standard reasoning agent that compresses history into natural language summaries. This represents the typical approach in existing folding reasoning systems.
- **SSF:** Our complete implementation that incorporates both task description and result summarization. The system first describes the sub-task goal via $\mathcal{V}_{judge}$, then provides the execution summary upon completion, creating a two-stage Executable Symbolic State: *Assistant*: "I'll create a branch to complete the subsequent reasoning. {describe}"; *User*: "Branch has finished its task, the returned message is: {summary} + {code}". This represents the full dual-pathway validation and context folding mechanism described in Section 3.

A.4 Implementation Details

For the semantic judge ($\mathcal{V}_{judge}$), we employ the same base model in a self-critique mode. The state rendering function $\mathcal{R}(\cdot)$ extracts variable assignments and verified lemmas while omitting intermediate retry logs.

B Prompt Engineering

B.1 SSF Reasoning Workflow

To implement the Branch-Execute-Fold protocol, our system prompt strictly enforces the generation of Executable Symbolic States (ESS) over natural language summaries. The prompt guides the agent to avoid "semantic drift" by grounding every reasoning step in a verified artifact (d_t) and its validation evidence (r_t).

You are a reasoning agent operating under the
 ↳ "Structured Strategy Folding" (SSF) paradigm.
 ↳ Your goal is to solve long-horizon problems
 ↳ by constructing a sequence of verified
 ↳ "Executable Symbolic States" (ESS).
 Do NOT rely on vague natural language summaries
 ↳ of your progress. Instead, you must
 ↳ rigorously validate and "fold" every step.

Current Folded Memory ($\$M_{\{t-1\}}$):
 {{ memory_context }}

Phase 1: Strategy & Decomposition

1. Analyze the Problem ($\$P$) and the current
 ↳ Folded Memory ($\$M_{\{t-1\}}$).
2. Identify the next immediate sub-task.

3. Determine the type of validation required:
 - * **Computation:** Requires ``execute_code``
 ↳ (e.g., arithmetic, algebra, simulation).
 - * **Logic:** Requires ``judge_lemma`` (e.g.,
 ↳ proving a geometric property, logical
 ↳ deduction).
4. If the sub-task requires multiple steps, use
 ↳ the ``branch`` tool to isolate the reasoning
 ↳ context.

Phase 2: Execution & Validation (Inside Branch)

1. Generate a candidate derivation ($\$d_t$).
2. **Validator Call:**
 - * If Computational: Write Python code. The
 ↳ output must be the exact numerical value
 ↳ or variable assignment.
 - * If Logical: Formulate a precise lemma.
 ↳ Use the judge to verify it follows
 ↳ strictly from $\$M_{\{t-1\}}$.
3. **Fail-Fast:** If validation fails (e.g., code
 ↳ error or logical rejection), refine $\$d_t$
 ↳ immediately within the branch.

Phase 3: Context Folding

1. Once a derivation is verified ($\$r_t$ is
 ↳ obtained), call ``fold_state``.
2. This will discard the temporary reasoning
 ↳ tokens and append only the verified ESS tuple
 ↳ $\$(d_t, r_t)$ to the global memory.
3. Treat folded states as immutable anchors.

<IMPORTANT>

* Never summarize a result using text like "I
 ↳ calculated x and it was 5". Fold the state
 ↳ ``x=5`` directly.
 * The ``description`` in branching must be a
 ↳ semantic tag (e.g., "Solve quadratic for t").
 * You must obtain a successful validation
 ↳ signal (exit code 0 or "Approved") before
 ↳ folding.
 </IMPORTANT>

B.2 Semantic Judge Prompt

To ensure the reliability of logical deductions, the Semantic Judge ($\mathcal{V}_{judge}$) evaluates candidate assertions against the accumulated verified memory. The prompt enforces strict logical entailment.

You are the Semantic Judge for a formal reasoning
 ↳ system.
 Your goal is to verify whether a "Candidate
 ↳ Lemma" is a valid logical consequence of the
 ↳ "Verified Context".

Verified Context ($\$M_{\{t-1\}}$):
 {{ memory_context }}

Candidate Lemma ($\$d_t$):
 {{ candidate_lemma }}

Task:

Determine if $\$d_t$ is necessarily true given
 ↳ $\$M_{\{t-1\}}$.
 * If it is a valid deduction, output "Status:
 ↳ APPROVED".
 * If it relies on unproven assumptions,
 ↳ contains a logical leap, or contradicts the
 ↳ context, output "Status: REJECTED" and
 ↳ explain the error.

Constraints:

- * Be rigorous. Do not accept "likely" or
 $\hookrightarrow$ "intuitive" steps; they must be logically
 $\hookrightarrow$ necessary.
- * Ignore any information not present in the
 $\hookrightarrow$ Verified Context or the original Problem
 $\hookrightarrow$ Statement.

C Agent Scaffold

C.1 State Rendering and Rollback

As distinct from standard context summarization methods like Sun et al. (2025), SSF does not compress history into natural language. When `fold_state` is invoked: 1. The Validator Check ensures the evidence matches the content. 2. The Context Rollback mechanism removes all intermediate tokens generated within the current branch scope from the KV-cache. 3. The State Rendering Function $R(s_t)$ injects the tuple $s_t = (d_t, r_t)$ into the prompt as a structured block, ensuring $O(1)$ context growth per reasoning step.

C.2 SSF Toolset

To enable the Dual-Pathway Validation mechanism, we implement a specific set of tools that separate generation from verification. Following the formulation in Equation 3, the agent interacts with the environment using the following definitions:

```
branch = {
  'type': 'function',
  'function': {
    'name': 'branch',
    'description': """Initiate a
 $\hookrightarrow$  sub-trajectory (branch) for a
 $\hookrightarrow$  specific sub-task.
This isolates the local reasoning context.
 $\hookrightarrow$  When the task is done, the branch is
 $\hookrightarrow$  closed and only the 'folded' result
 $\hookrightarrow$  is returned to the main memory.""",
    'parameters': {
      'type': 'object',
      'properties': {
        'description': {
          'type': 'string',
          'description': 'A semantic tag
 $\hookrightarrow$  describing the sub-task
 $\hookrightarrow$  (e.g., "Calculate
 $\hookrightarrow$  discriminant", "Derive
 $\hookrightarrow$  Lemma 2").',
        },
      },
      'prompt': {
        'type': 'string',
        'description': 'The specific
 $\hookrightarrow$  constraints and initial
 $\hookrightarrow$  variables passed to the
 $\hookrightarrow$  sub-agent.',
      },
    },
  },
  'required': ['description', 'prompt'],
},
```

```
},
}

execute_code = {
  'type': 'function',
  'function': {
    'name': 'execute_code',
    'description': """The Deterministic
 $\hookrightarrow$  Executor (V_exec).
Executes Python code in a sandboxed
 $\hookrightarrow$  environment to verify computational
 $\hookrightarrow$  derivations.
Returns the standard output and the final
 $\hookrightarrow$  state of defined variables.""",
    'parameters': {
      'type': 'object',
      'properties': {
        'code': {
          'type': 'string',
          'description': 'Valid Python
 $\hookrightarrow$  code block. Must print
 $\hookrightarrow$  the final result.',
        },
        'timeout': {
          'type': 'integer',
          'description': 'Execution
 $\hookrightarrow$  timeout in seconds
 $\hookrightarrow$  (default: 10).',
        },
      },
    },
    'required': ['code'],
  },
},

judge_lemma = {
  'type': 'function',
  'function': {
    'name': 'judge_lemma',
    'description': """The Semantic Judge
 $\hookrightarrow$  (V_judge).
Verifies logical assertions that cannot
 $\hookrightarrow$  be executed as code.
Checks if the assertion follows logically
 $\hookrightarrow$  from the previous Folded Memory.""",
    'parameters': {
      'type': 'object',
      'properties': {
        'assertion': {
          'type': 'string',
          'description': 'The logical
 $\hookrightarrow$  statement or lemma to be
 $\hookrightarrow$  verified.',
        },
        'premise_ids': {
          'type': 'array',
          'items': {'type': 'integer'},
          'description': 'List of
 $\hookrightarrow$  previous ESS IDs used as
 $\hookrightarrow$  premises for this
 $\hookrightarrow$  deduction.',
        },
      },
    },
    'required': ['assertion'],
  },
},

fold_state = {
  'type': 'function',
```

```

'function': {
  'name': 'fold_state',
  'description': """Commits a verified
↳ Executable Symbolic State (ESS) to
↳ memory.
Calling this triggers the State Rendering
↳ Function R(s_t) and rolls back the
↳ context window, keeping only this
↳ artifact."""",
  'parameters': {
    'type': 'object',
    'properties': {
      'type': {
        'type': 'string',
        'enum': ['computational',
↳ 'logical'],
        'description': 'The type of
↳ the verified state.',
      },
      'content': {
        'type': 'string',
        'description': 'The core
↳ artifact (e.g., variable
↳ assignment "x=5" or
↳ lemma "Triangle ABC is
↳ isosceles").',
      },
      'evidence': {
        'type': 'string',
        'description': 'The output
↳ from V_exec or V_judge
↳ that proves this content
↳ is valid.',
      },
    },
    'required': ['type', 'content',
↳ 'evidence'],
  },
},
}

```

D Case Study: Complete SSF Trajectory

To illustrate the effectiveness of SSF, we present a complete case study demonstrating the Branch-Execute-Fold protocol in action. The problem is a mathematical reasoning task requiring multi-step derivation and verification.

D.1 Problem Statement

“Calculate the interval for the second bell: Four bells toll together at intervals of 9, x , 14, and 18 seconds respectively. If they toll together again after 630 seconds, find the missing interval x .” This problem requires identifying the least common multiple (LCM) relationship and solving for the unknown.

D.2 SSF Execution Trace

Main Reasoning Thread (Planning State): The agent begins in the main thread, observing the problem and the empty folded memory $\mathcal{M}_0 = []$. It decomposes the task:

1. Compute LCM of known intervals (9, 14, 18)
2. Determine the missing factor to reach LCM = 630
3. Identify candidates for x
4. Verify the smallest candidate

Step 1: Procedural Branching (Execution State) The agent recognizes that computing the LCM is a computational sub-task. It invokes:

```
branch(description="Compute LCM of known intervals",
        prompt="Calculate lcm(9, 14, 18)")
```

Step 2: Execution and Validation within Branch Inside the branch, the agent generates a candidate derivation d_1 (Python code). However, it initially attempts to pass a list directly to `math.lcm`, causing a type error:

```
import math
intervals = [9, 14, 18]
# Incorrect usage: passing list instead of unpacked arguments
lcm_known = math.lcm(intervals)
print(f"LCM(9,14,18) = {lcm_known}")
```

The Deterministic Executor $\mathcal{V}_{exec}$ executes this code:

- **Execution Result:** `TypeError: 'list' object cannot be interpreted as an integer`
- **Exit Code:** 1 (failure)
- **Validation:** FAIL

Self-Correction Loop: As validation fails, the system triggers a **Self-Correction Loop**, prompting $\mathcal{G}$ to regenerate d_t based on the feedback, preventing error propagation. The agent refines the code to correctly compute LCM iteratively:

```
import math
intervals = [9, 14, 18]
lcm_known = 1
for v in intervals:
    lcm_known = lcm_known * v // math.gcd(lcm_known, v)
print(f"LCM(9,14,18) = {lcm_known}")
```

The Deterministic Executor $\mathcal{V}_{exec}$ re-executes this code:

- **Execution Result:** $\text{LCM}(9, 14, 18) = 126$
- **Exit Code:** 0 (success)
- **Validation:** PASS

Step 3: State Folding (Commit Step) Upon successful validation, the agent folds this result into an ESS:

$$s_1 = (d_1, r_1) = (\text{"Compute LCM of known intervals"}, \text{"LCM}(9, 14, 18) = 126\text{"})$$

The folded memory is updated: $\mathcal{M}_1 = [s_1]$.

The entire sub-trajectory (intermediate reasoning steps) within the branch is discarded from the KV-cache. Only the compact rendered state $\mathcal{R}(s_1)$ is retained in the main thread:

Verified Computation: $\text{LCM}(9, 14, 18) = 126$

Back to Main Thread (Planning State) The agent resumes in the main thread with the folded memory $\mathcal{M}_1$. It observes the verified result ($\text{LCM} = 126$) and notes that the target LCM is 630. It reasons: $630 = 126 \times 5$, so the missing interval x must introduce factor 5. It branches again:

```
branch(description="Find all candidates for x",  
        prompt="Find divisors of 630 that contain factor 5")
```

Step 4: Second Branch Execution Inside the second branch, the agent generates d_2 :

```
candidates = []  
for a in [0,1]:  
    for b in [0,1,2]:  
        for d in [0,1]:  
            val = 5 * (2**a) * (3**b) * (7**d)  
            candidates.append(val)  
print(f"Candidates: {sorted(candidates)}")
```

Execution result: Candidates: [5, 10, 15, 30, 35, 45, 70, 90, 105, 210, 315, 630]

Validation: PASS. Folded into s_2 , $\mathcal{M}_2 = [s_1, s_2]$.

Step 5: Third Branch - Verification The agent branches to verify the smallest candidate:

```
import math  
x = 5  
lcm_all = math.lcm(9, x, 14, 18)  
print(f"lcm(9, {x}, 14, 18) = {lcm_all}")  
assert lcm_all == 630, f"Expected 630, got {lcm_all}"
```

Execution result: $\text{lcm}(9, 5, 14, 18) = 630$. Validation: PASS. Folded into s_3 , $\mathcal{M}_3 = [s_1, s_2, s_3]$.

D.3 Key Advantages Demonstrated

1. Lossless Context Compression: Instead of retaining verbose intermediate reasoning (e.g., trial-and-error attempts, debugging traces), SSF retains only three compact, verified states. The rendered memory is:

Verified Computation: $\text{LCM}(9, 14, 18) = 126$

Verified Computation: Candidates = [5, 10, 15, ...]

Verified Computation: $\text{lcm}(9, 5, 14, 18) = 630$

2. Error Prevention: If the agent had generated an incorrect intermediate result (e.g., computing LCM incorrectly as 100 instead of 126), the validator would reject it, triggering a self-correction loop. The main thread never sees unverified information.

3. Bounded Context Growth: The active context size remains constant across steps: it contains only the problem statement and three folded states, regardless of how many intermediate computations occurred within the branches. This prevents the “lost-in-the-middle” phenomenon.

4. Reproducibility and Auditability: Each folded state is reproducible: the agent can re-execute d_i to verify r_i . This ensures the reasoning chain is fully traceable and verifiable, eliminating semantic drift from linguistic approximations.