A Neuro-Symbolic Multi-Agent Framework for Complex Temporal Reasoning in UnSeenTimeQA

Anonymous ACL submission

Abstract

Despite the strong performances of large language models (LLMs) on the temporal reasoning task, recent studies using the Unseen-TimeQA benchmark indicates that this proficiency may stem largely from data contamination—specifically, memorizing test-set facts during pre-training. Notably, even advanced LLMs like DeepSeek-V3 struggle on this benchmark. To address this limitation, we propose NSMATeR, a neuro-symbolic multi-agent framework that formalizes the error-prone reasoning steps into executable code. By integrating this symbolic component, our approach effectively mitigates LLM failures, such as event forgetting, rule misunderstanding and hallucination. To ensure the code correctness, we further introduce a case-verification mechanism that iteratively refines the code using a small set of annotated cases. Extensive experiments on UnseenTimeQA show that our framework significantly boosts accuracy, improving even DeepSeek-V3's performance by 27.89% in the most challenging scenarios.

1 Introduction

Temporal reasoning serves as a crucial evaluation for large language models (LLMs) (Piryani et al., 2025). For instance, when asked questions like "*What citizenship did Vladimir Bukovsky hold between Jun 1990 and Dec 1990?*", an LLM must infer the target person's status during the specified time period from the given documents.

Although prevailing LLMs, such as GPT-4o (OpenAI et al., 2024) and DeepSeek (DeepSeek-AI, 2024), demonstrate outstanding performance on temporal reasoning tasks (Chu et al., 2024), recent studies indicate that these capabilities likely stem from data contamination (Uddin et al., 2025). Specifically, during pre-training, models may learn facts from data collected before their release. Consequently, they often rely on memorization rather than logical reasoning to answer temporal questions.

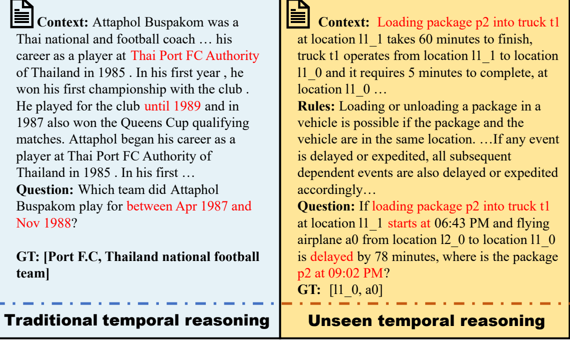

Figure 1: The left panel shows an example from TSQA, where the model reason from the factual context to answer the question. The right panel presents an example from UnseenTimeQA, where the question is built on a sequence of artificial events. The model must understand and apply the predefined temporal rules to perform inference.

However, memory-independent temporal reasoning is an indispensable capability for LLMs. Practical applications, such as arranging travel plans, require models to integrate dynamic, real-time conditions—like diverse destinations and variable transportation schedules—information typically absent from their pre-training data. Thus, to rigorously evaluate this capability, Uddin et al. (2025) recently proposed the UnseenTimeQA benchmark. It dynamically generates temporal questions by constructing artificial, unseen events, forcing models to rely on logical reasoning rather than memorization. Figure 1 illustrates the difference between traditional and unseen temporal reasoning.

So far, even advanced large-scale LLMs like GPT-4o perform poorly on this challenging benchmark, achieving only 42% accuracy (Uddin et al., 2025). Our error analysis identifies three primary failure modes: (1) **Event Forgetting** (25% of errors): Given lengthy sequences (averaging 30 events), models frequently overlook events during

reasoning. (2) **Rule Misunderstanding** (70% of errors): Models often produce reasoning steps that violate predefined rules, such as conflating parallel and sequential events. (3) **Hallucination**: Inherent to LLMs, this involves generating content not grounded in the provided event sequence. Crucially, we observe that the key temporal reasoning step, i.e., inferring the start and end times of events, are perfectly suited for symbolic reasoning. This is due to its deterministic nature and the need to strictly adhere to predefined temporal rules. Therefore, a correct symbolic procedure can supplement or replace the LLM's reasoning to eliminate errors like event forgetting and rule misunderstanding.

Motivated by these insights, we propose a neuro-symbolic multi-agent framework for this unseen temporal reasoning task. The core of our framework is to formalize the key reasoning step, i.e., predicting the timeline of an event sequence, into executable code. The framework employs two cooperative agents: a symbolic agent that executes the code for deterministic reasoning, and a neural agent that generates the code, prepares the formatted inputs, and synthesizes the final answer. This design leverages the complementary strengths of both paradigms. The symbolic component provides rigorous, rule-based reasoning to counter inherent LLM weaknesses. Conversely, the neural component overcomes the rigidity of symbolic component by handling natural language inputs and outputs.

Since the generated code performs the core reasoning, ensuring its correctness is crucial. However, producing flawless code under complex temporal rules remains challenging, even for state-of-the-art LLMs. To overcome this, we further introduce a case-verification mechanism that uses a small set of annotated cases to iteratively refine the generated code, ensuring it matches each rule.

Our contributions can be summarized as follows:

1. We propose a neuro-symbolic multi-agent framework for the UnseenTimeQA task. By integrating a symbolic component, our approach effectively mitigates the issues of event forgetting, rule misunderstanding and hallucination inherent in LLM-based reasoning.

2. We introduce a case-verification mechanism that iteratively refines the code generation using a small set of cases to ensure its correctness and robustness.

3. Extensive experiments show our framework significantly boosts the performance of SOTA models (e.g. GPT-4o, DeepSeek-v3), achieving an accuracy gain of up to 27.89% in the most complex scenarios.

2 Related Work

2.1 LLMs for Temporal Reasoning

With the advancement of LLMs, researchers have recently focused on evaluating their performance on temporal reasoning tasks. Current research primarily falls into two lines: traditional and unseen temporal reasoning.

Traditional Temporal Reasoning This line evaluates LLMs on answering temporal questions about factual events, where models are provided with a context (e.g., historical events) and reason directly from it to answer questions. For instance, TG-LLM (Xiong et al., 2024) introduced a two-stage method that first converts natural language into a temporal graph before performing inference. CoTempQA (Su et al., 2024) presented a math-reasoning Chain-of-Thought approach to guide models in solving temporal problems. CoTime (Wu and Hooi, 2025) organizes events into an SQL schema to facilitate deduction from structured data. TempCoT (Song et al., 2025) proposed a three-stage strategy involving constraint extraction, semantic fact retrieval, and a temporal logic reasoning module to enhance model performance.

Unseen Temporal Reasoning Recent studies suggest that the strong performance of LLMs on traditional temporal reasoning tasks may be largely attributed to data contamination. To evaluate pure reasoning ability, Uddin et al. (2025) recently introduced a new benchmark, UnseenTimeQA. This benchmark generates questions based on artificial, unseen events, forcing models to reason from provided rules rather than memorization. Research in this benchmark remains nascent due to its novelty. To our knowledge, RASTeR (Schumacher et al., 2025) is the sole and most recent work, proposing a prompting framework that decouples context evaluation from answer generation. It first assesses the provided context before constructing a temporal knowledge graph to guide reasoning. Nevertheless, its suboptimal performance underscores the persistent challenge of achieving robust unseen temporal reasoning.

2.2 Neuro-Symbolic Methods

Existing research demonstrates that integrating symbolic components is effective to enhance the LLMs' reasoning capabilities (Bhuyan et al., 2024). For instance, Logic-LM (Pan et al., 2023) pioneered the incorporation of symbolic agents into LLM reasoning pipelines, improving accuracy in logical reasoning tasks. Similarly, PEIRCE (Quan et al., 2025) introduced a neuro-symbolic framework, unifying different reasoning modes through an iterative conjecture-criticism process. In the domain of temporal reasoning, TReMu (Ge et al., 2025) applied a neuro-symbolic approach, generating executable programs to replace the LLMs' step-by-step reasoning. However, existing works has yet to address the unseen temporal reasoning task, which demands a precise understanding and application of the predefined temporal rules. Moreover, generating symbolic procedures that strictly match these intricate rules remains a challenge.

2.3 Code-based Methods

Code-based methods aim to translate problems suitable for programming solutions into executable code (Xu et al., 2025). For example, POT (Chen et al., 2022) first integrated the programming-execution process into the chain of thought. PAL (Gao et al., 2023) decouples code generation from the overall problem-solving steps by delegating the programming task to a separate LLM. To improve the correctness of generated code, CODEPLAN (Lu et al., 2025) constructed a new mathematical dataset that pairs natural language reasoning steps with corresponding code. However, the majority of existing work focuses on mathematical problems. Translating these problems into code is relatively straightforward due to their formalized rules and operations. In contrast, unseen temporal reasoning involves natural language rules that may lie beyond the model's pre-training data, posing a significant challenge for code translation.

3 Problem Definition

Formally, given a set of rules R in natural language, an event sequence $E = \{e_1, ..., e_n\}$ containing n events and a question Q, the model must apply R to infer the timing of events in E and ultimately answer Q. Specifically, each event e_i is described in natural language, specifying only its duration, while its start and end times are not provided. The question Q provides the occurrence time of one event in E, along with an expedited or postponed time for another event. Using this information, the model is to determine the start and end times of all other events to arrive at final answer.

4 Method

Our neuro-symbolic multi-agent framework consists of two steps: Code Generation and Neuro-Symbolic Reasoning. First, in Code Generation, a neural agent (the code generator) generate executable code based on predefined rules to deduce event start and end times. A second neural agent (the debugger) then refines this code through iterative verification against several cases. Once the code is finalized, Neuro-Symbolic Reasoning is performed to answer the question. An overview of the framework is illustrated in Figure 2.

4.1 Code Generation

The core of this temporal reasoning task is to predict the timeline of an event sequence, i.e., leveraging information about one event's expedited or postponed time to deduce the start and end times of all other events. This is achieved in code through an initial generation followed by case-verification.

4.1.1 Initial Code Generation

The code generator is prompted to produce initial code based on the predefined rules R. To clarify the task, we also provide the generator with a few demonstrations. Each demonstration includes three components: an unannotated event sequence E' as input, a condition H' that provides expedited or postponed time of one event, and the correct output—a fully annotated sequence $\bar{E}'$ where all start and end times are deduced from that given condition. This process can be formalized as follows:

$$C^{(0)} = \text{LLM}_G(\mathcal{P}_\text{G} \oplus R \oplus D), \quad (1)$$

where $\mathcal{P}_\text{G}$ denotes the code generation prompt, $D = \{H'_1, (E'_1, ..., E'_n), (\bar{E}'_1, ..., \bar{E}'_n)\}$ denotes the demonstration set containing n examples, and $C^{(0)}$ denotes the initially generated code. For processing, the data is formatted as follows: each event in E'_i is a tuple (action, subject, object, location, duration); each event in $\bar{E}'_i$ is a tuple (start time, event string, end time); and each condition H'_i is a tuple (action, expedited/postponed, subject, object, location, expedited/postponed time).

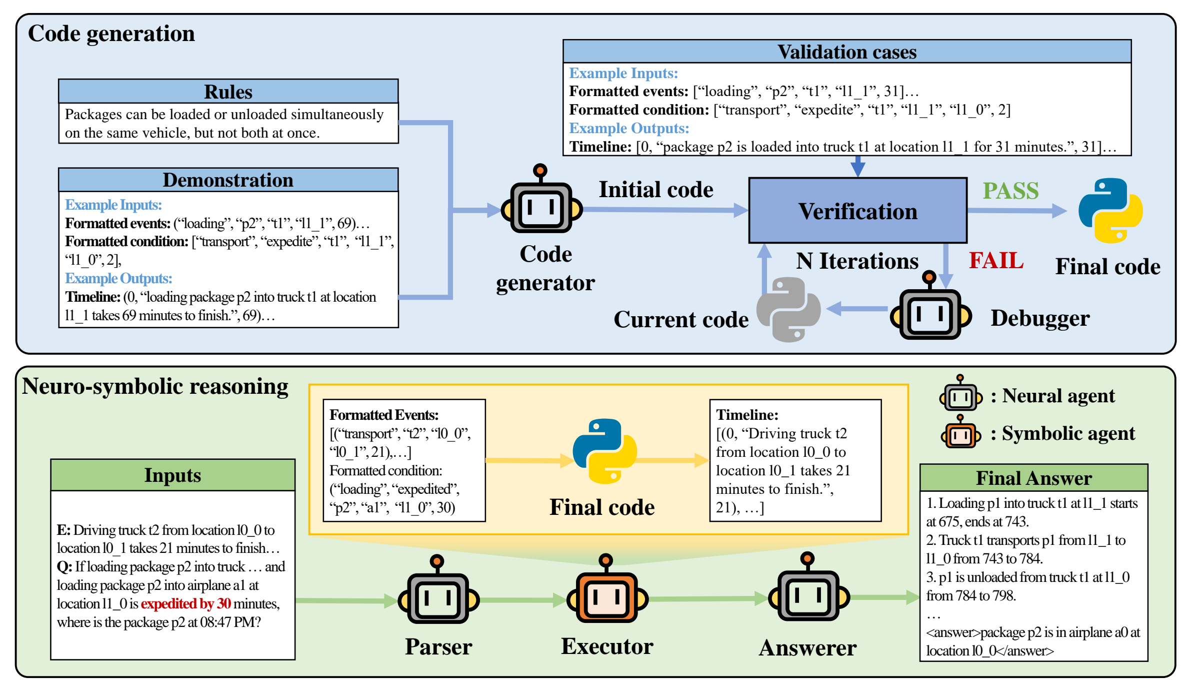

Figure 2: The overview of our neuro-symbolic multi-agent framework.

4.1.2 Case-Verification Mechanism

Code generation often suffers from rule misunderstanding, particularly when rules are complex. Inspired by human debugging practices, we introduce case-verification to refine the initial code.

First, we construct a verification set $V = [\{H'_i, (E'_1, .., E'_n), (\bar{E}'_1, .., \bar{E}'_n)\}], i \in [1, m]$ containing m cases. These cases are manually selected to cover all predefined rules. The debugger agent executes the code on each unannotated sequence E'_i, predicts timeline $\hat{E}'_i$. This prediction is compared against the ground-truth $\bar{E}'_i$. If a mismatch occurs, the debugger receives the rules R, the current code $C^{(i)}$, $\hat{E}'_i$, H'_i and $\bar{E}'_i$, and is prompted to analyze the errors and produce a revised code. The new code is then validated again using V. This loop repeats iteratively until the code's outputs match the annotated timelines for all cases in V. The case-verification process can be formalized as follows:

$$f_{\text{CASE_VERIFICATION}}(V, C^{(t)}) = \begin{cases} C^{(t)}, & \text{if } \forall i \in \{1, \dots, m\},\ \hat{E}'_i = \bar{E}'_i \\ f_{\text{CASE_VERIFICATION}}(V, C^{(t+1)}), & \text{otherwise} \end{cases} \quad (4)$$

where t is the iteration index, and

$$C^{(t+1)} = f_{\text{DEBUGGER}}(R, C^{(t)}, \hat{E}'_i, H'_i, \bar{E}'_i), \quad \text{if } \hat{E}'_i \neq \bar{E}'_i$$

Through a few iterations, the debugger searches in the code space to identify the optimal solution that can pass all cases. This verifies that the code logic may be consistent with the given rules, thereby guaranteeing its accuracy and ensuring the success of the overall framework.

4.2 Neuro-Symbolic Reasoning

To this end, we employ the final code to conduct neuro-symbolic reasoning. First, a neural parser converts the raw natural language inputs—the event sequence and the question—into the formatted data required by the code. Next, a symbolic agent executes the code to generate the timeline of the event sequence. Finally, a neural answerer integrates this timeline with the question to produce the final answer. Note that the timeline produced by the code is relative, starting from zero o'clock. The answerer converts this to an absolute timeline by applying a temporal offset, which is derived from the occurrence time of a specific event provided in the question.

5 Experiments

5.1 Setup

Dataset We evaluate our method on the UnseenTimeQA benchmark (Uddin et al., 2025). The tasks are categorized into three distinct difficulty levels: Easy, Medium, and Hard. Since advanced LLMs

Model	Serial				Parallel			
	Static-Time	Relative-Time	Hypothetical-Time	Average	Static-Time	Relative-Time	Hypothetical-Time	Average
Gemma-2-27B	13.00 (1.85)	14.66 (0.66)	17.77 (0.77)	15.14	12.99 (1.20)	12.99 (2.90)	15.10 (2.34)	13.69
RASTeR	-	-	-	35.5	-	-	-	39.1
Llama-3.1-70B	41.50 (1.64)	40.00 (0.47)	33.66 (0.94)	38.38	42.50 (2.12)	36.16 (2.12)	40.33 (0.94)	39.66
GPT-4o	57.11 (1.57)	47.44 (2.87)	44.77 (3.01)	49.77	47.33 (2.60)	39.11 (2.98)	42.11 (1.83)	42.85
DeepSeek-V3	46.56 (2.22)	41.33 (1.19)	40.11 (1.40)	42.67	59.78 (3.03)	57.11 (2.47)	58.44 (3.24)	58.44
Ours w/ DeepSeek-V3	**92.00 (1.70)**	72.67 (3.14)	70.36 (1.57)	78.34	85.44 (1.93)	75.56 (1.13)	69.78 (2.94)	77.93
Ours w/ GPT-4.1	90.00 (1.24)	**83.00 (2.38)**	**87.00 (1.11)**	**86.67**	**88.33 (0.87)**	**84.33 (1.17)**	**86.33 (1.85)**	**86.33**
Human	100	93.33	86.66	93.33	93.33	86.66	73.33	84.44

Table 1: Accuracy comparison on the UnseenTimeQA dataset under the Hard setting, with the best and second-best results highlighted in bold and underline, respectively. Average results from three independent runs are reported, with standard deviations in parentheses.

such as GPT-4o already achieve high accuracy (exceeding 90%) on the Easy and Medium levels, we focus our experiments on the Hard setting. A detailed description of Hard setting dataset is provided in Appendix A.

Implementation Details We evaluate our framework using two primary LLM backbones: DeepSeek and GPT. For DeepSeek, we employ DeepSeek-R1 as the code generator and debugger in the Code Generation, while DeepSeek-V3 serves as the neural agent for natural language processing. For GPT, we utilize GPT-o3 for code generation and debugging, and GPT-4.1 as the neural agent for natural language processing. The selection of DeepSeek-R1 and GPT-o3 is based on their superior code-specific capabilities.

Baselines Baseline models include the most recent work, RASTeR (Schumacher et al., 2025), on this benchmark. For comparison, we also consider several LLMs, specifically Gemma-2-27B, Llama-3.1-70B, GPT-4o, and DeepSeek-V3—all of which employ chain-of-thought prompting for this task. Some of results are retrieved from (Uddin et al., 2025).

5.2 Main Results

Performance comparison on the UnseenTimeQA benchmark is presented in Table 1. We make several key observations.

First, our experiments show that a single LLM utilizing chain-of-thought reasoning struggles to effectively solve the UnseenTimeQA problem. For example, both Llama-70B and the large-scale model DeepSeek-V3 achieve only 50% accuracy. While the RASTer method improves performance through context filtering, its results remain relatively low.

In contrast, our proposed framework demonstrates a significant advantage over all existing baselines. It achieves an average accuracy improvement of 36.9% over the second-best method on the Serial dataset, and 27.89% on the Parallel dataset. These results confirm the efficacy of our framework in enhancing temporal reasoning capabilities. Notably, our method even exceeds human-level performance in certain scenarios, indicating that for complex multi-step temporal reasoning, the logically sound symbolic component can replace humans to perform stable and accurate reasoning.

Furthermore, the consistent performance gains observed with both DeepSeek-V3 and GPT-4.1 as inference backbone verify that our framework is effective across different underlying model architectures. Finally, with the robust temporal reasoning provided by the symbolic component, our method maintains high accuracy (above 70%) consistently across scenarios of varying difficulty.

5.3 Ablation Study

To evaluate the contribution of key components in our framework, we conduct an ablation study using several variants. Due to resource constraints, these experiments are performed using DeepSeek-V3 as the base model. The ablated methods are defined as follows:

- **w/o Neural**: This variant removes all neural components, resulting in a purely code-based approach. Specifically, the model is prompted to generate executable code that directly outputs the answer.
- **w/o Symbolic**: This variant removes all symbolic components, resulting in a purely neural-based method. The model is directly prompted to solve the question using the provided context and rules. To ensure a fair com-

Variants	Serial				Parallel			
	Static-Time	Relative-Time	Hypothetical-Time	Average	Static-Time	Relative-Time	Hypothetical-Time	Average
w/o Neural	31.33(2.09)	26.00(1.46)	26.67 (3.89)	28.00	23.00 (3.27)	28.33 (3.40)	20.67 (2.36)	24.00
w/o Symbolic	46.44(1.49)	39.33(2.62)	34.11(2.19)	39.96	56.89 (1.59)	52.67 (1.19)	50.56 (3.71)	53.37
w/o Case-verification	92.00 (1.70)	72.67 (3.14)	70.36 (1.57)	78.34	77.33 (3.14)	65.44 (3.94)	22.77 (2.04)	55.18
Ours	92.00 (1.70)	72.67 (3.14)	70.36 (1.57)	78.34	85.44 (1.93)	75.56 (1.13)	69.78 (2.94)	77.93

Table 2: Results of the ablation study evaluating key components of our framework, using DeepSeek-V3 as backbone.

parison, we employ few-shot prompting with the same number of annotated cases used in the validation set of our framework.

- **w/o Case-verification**: This variant removes the case-verification mechanism from our framework while retaining all other components.

The ablation results are presented in Table 2. The "w/o Neural" model achieves an accuracy of only around 20%. This demonstrates that in complex scenarios, prompting the model to generate executable code for direct problem-solving is ineffective. The model often fails to produce syntactically or logically correct code.

The results for the "w/o Symbolic" model indicate that incorporating few-shot prompting does not enhance reasoning performance; in fact, it performs worse than zero-shot chain-of-thought reasoning (see Table 1). A possible reason is that the original input context is already substantial. Since each demonstration itself contains a lengthy token sequence, adding multiple demonstrations further extends the context. This increased length heightens the risk of event forgetting, ultimately degrading performance.

"w/o Case-verification" achieves strong performance in Serial scenarios, matching the results of our full method. This is because the underlying logical rules in Serial scenarios are relatively straightforward, allowing the model to frequently generate correct code in a single attempt that satisfies all validation cases. However, as the complexity increases in Parallel scenarios, the performance of this variant declines significantly. This sharp drop underscores the critical role of the case-verification mechanism, which becomes essential for ensuring high-quality code generation.

5.4 Impact of Case-Verification

To study the effectiveness of the case-verification mechanism, which iteratively refines the generated code, we measured the performance of the code produced in each round on the validation set. The accuracy of output timeline (measured on event level) was used as the evaluation metric, with the results shown in Figure 3.

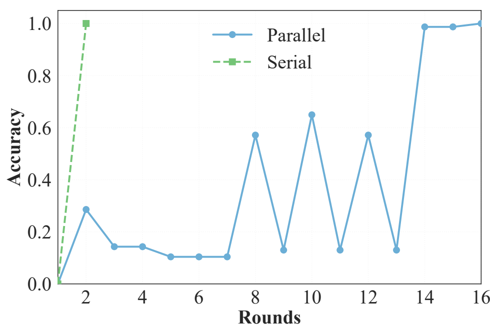

Figure 3: Timeline accuracy over successive case-verification iterations for the Parallel and Serial datasets.

As shown in Figure 3, the model successfully generates code that passes all validation cases in a single iteration for Serial rules. In contrast, for the more complex Parallel rules, the initial code passes only a small fraction of cases. This disparity suggests that complex logical structures require further refinement.

Furthermore, the accuracy of the generated code fluctuates during the initial debugging phases (from the first to the 12-th iteration) before demonstrating a clear upward trend. Through this iterative process, the model eventually produces a logical representation that matches all validation cases, proving the efficacy of our case-verification mechanism.

5.5 Impact of Different Validation Cases

As discussed, the case-verification mechanism is crucial for the performance of our framework on the Parallel dataset. While we used a manually constructed validation set to refine the generated code, such careful curation may not be feasible in all real-world scenarios. To assess robustness under less controlled conditions, we conducted an additional evaluation using validation cases ran-

domly selected from the dataset, with the temperature constraint of the code generator disabled. We report the performance of the generated code using the DeepSeek-V3 backbone on the Parallel dataset, with results detailed in Table 3.

	Static-Time	Relative-Time	Hypothetical-Time	Average
Random 1	82.67	72.67	76.67	77.34
Random 2	85.67	73.33	76.67	78.56
Random 3	83.00	73.33	74.33	76.89
Manual	85.44	75.56	69.78	77.93

Table 3: Performance evaluation (i.e., accuracy) using different validation case sets on the Parallel dataset.

As shown in Table 3, three randomly selected validation sets also yield comparable performance with manual selected cases. These results demonstrate that our framework remains effective even when validation cases are chosen at random.

5.6 Error Analysis

To better understand the performance improvements of our framework, we conducted an extensive error analysis comparing our method with a pure neural-based baseline (directly prompting an LLM for chain-of-thought reasoning) on the Parallel dataset, using DeepSeek-V3 as the backbone model. Specifically, we randomly sampled 60 erroneous cases from all errors made by the baseline DeepSeek-V3 method and performed manual error classification. The errors were categorized into three primary types: event forgetting, rule misunderstanding, and hallucination. The distribution of cases across these error types is presented in Figure 4.

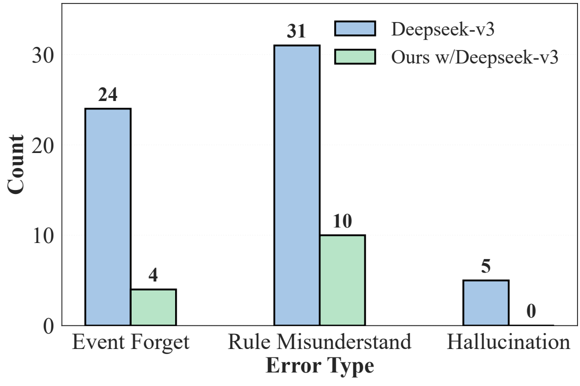

Figure 4: Error distribution comparison between the baseline DeepSeek-V3 model and our framework.

As shown in Figure 4, rule misunderstanding constitutes the most frequent error type, highlighting a persistent bottleneck in the model's ability to understand complex temporal logic. Event forgetting is the second most common issue, indicating challenges in maintaining consistent reasoning over long event sequences. Notably, hallucination occurs infrequently, appearing in only 5 of the 60 sampled cases. Critically, our framework demonstrates substantial error reduction across all three categories. This indicates that the same underlying model, when equipped with the symbolic component, can effectively overcome these core reasoning limitations.

To investigate the remaining performance bottlenecks, we conducted an error analysis on our framework's failures. From all incorrect predictions, we randomly sampled 86 cases for manual inspection. These errors were categorized into three main types: parsing errors, reasoning errors and answer errors. The breakdown is as follows: parsing errors constitute 51.16% of the total, reasoning errors account for 46.51%, and answer errors represent the remaining 2.33%.

Parsing errors constitute the most prevalent failure mode in our framework, representing 51.16% of all erroneous cases. This type of error occurs when the parser neural agent fails to produce a valid structured output (e.g., returning None), which immediately halts the entire reasoning pipeline. These failures can stem from various sources, such as network request issues or malformed output generation, reflecting inherent limitations in the reliability of neural agents.

Reasoning errors represent the second most common failure type in our framework. These errors occur when the neural agent successfully parses and outputs valid data, but the timeline subsequently generated by the code deviates from the ground truth. We analyze this category below with a representative example (see Figure 5).

Code output:
[…, (862, 'unloading package p2 fromairplane a1 at location l0_0 takes 90 minutesto finish.', 952), …]
Format question: If loading package p2 intotruck t1 at location l1_3 starts at 632. Whereis the package p2 at 874?
Answer: l0_0
Gt: a1

Figure 5: A case of reasoning error.

In this case, the final answer is consistent with

the output timeline, yet differ from the ground truth. The discrepancy arises because the timeline produced by the generated code is incorrect. Specifically, in the correct timeline, the unloading operation should have been completed by the given timestamp. This type of error originates from partial logical flaws in the generated code. Since natural language descriptions of rules cannot exhaustively cover every detail, and the validation cases used in the case-verification mechanism cannot guarantee full coverage of all branching scenarios, the code fails to accurately capture the rules for some edge cases. However, this type of error is addressable: by analyzing failure cases, selecting representative misclassified samples, and adding them to the validation set, the corresponding code can be refined through further iterations of the case-verification process.

Answer errors are the least frequent failure type. As shown in Figure 6, their defining characteristic is that the code generates a correct timeline, yet the final answer produced by the neural answerer agent—based on this timeline—is wrong. The cause of such errors is relatively specific, primarily stemming from the answerer agent's failure to correctly locate or interpret the required information within the timeline.

Code output:
[…, (749, 'package p1 unloaded from airplane a0 at location l1_0 for 57 minutes.', 806), …]
Format question: If loading package p1 into truck t0 at location l0_1 starts at 363. Where is the package p1 at 788?
Answer: l2_0
Gt: l1_0, a0

Figure 6: A case of answer error.

In summary, parsing errors and answer errors together account for approximately 53% of all failures, both stemming from the inherent instability in the neural agent's generation process. This represents the primary performance bottleneck of our current framework. In contrast, the reasoning errors, which constitute 46.51% of cases, are more tractable and can be effectively addressed by expanding the validation case set to cover a wider range of cases.

5.7 Inference Cost Comparison

To evaluate the runtime efficiency of our method, we measured the average execution time and token consumption per sample on the Parallel dataset. The results are summarized in Table 4.

As the data indicate, our framework incurs a modest increase in inference time and token usage compared to a single-pass LLM method. This overhead stems from the multiple calls to neural agents required for different subtasks within our pipeline. In comparison to the deep-thinking model DeepSeek-R1, our approach demonstrates significantly lower costs in both execution time and token consumption.

Model	Question type	Times(s)	Tokens
DeepSeek-v3	Static	21.76	1976.7
	Relative	18.29	1935.8
	Hypothetical	23.44	1982.8
Ours w/ DeepSeek-v3	Static	94.04	6944.9
	Relative	88.17	7488.7
	Hypothetical	88.74	7346.8
GPT 4.1	Static	13.41	2245.6
	Relative	22.11	2181.6
	Hypothetical	23.31	2543.5
Ours w/ GPT 4.1	Static	39.86	6761.0
	Relative	41.02	7373.6
	Hypothetical	37.16	6868.6
DeepSeek-R1	Static	645.45	18181.8
	Relative	626.37	16216.2
	Hypothetical	997.06	24537.6

Table 4: Average inference time and token consumption per sample on the Parallel dataset across different methods.

6 Conclusion

In this paper, we introduce NSMATeR, a neuro-symbolic multi-agent framework to address the UnseenTimeQA problem. Our approach harnesses the code generation capabilities of large language models (LLMs) to transform complex temporal reasoning steps into executable code. To ensure the robustness of the generated code, we further propose a case-verification mechanism that iteratively refines the code using few annotated cases. Through extensive experiments, we demonstrate that our framework significantly outperforms advanced models like DeepSeek-V3 and GPT-4o up to 27.89% accuracy. Ablation study result validate the effectiveness of each core module within our architecture and underscore the advantage of integrating symbolic reasoning with neural language understanding for complex temporal reasoning tasks.

7 Limitation

Our framework has only been tested on the UnseenTimeQA task, future research can extend its application to other reasoning tasks with complex rules. Furthermore, although our framework has significantly improved reasoning accuracy, the overhead incurred by invoking multiple agents has increased the overall inference cost.

References

Bikram Pratim Bhuyan, Amar Ramdane-Cherif, Ravi Tomar, and TP Singh. 2024. Neuro-symbolic artificial intelligence: a survey. *Neural Computing and Applications*, 36(21):12809–12844.

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*.

Zheng Chu, Jingchang Chen, Qianglong Chen, Weijiang Yu, Haotian Wang, Ming Liu, and Bing Qin. 2024. TimeBench: A comprehensive evaluation of temporal reasoning abilities in large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1204–1228, Bangkok, Thailand. Association for Computational Linguistics.

DeepSeek-AI. 2024. Deepseek-v3 technical report. *Preprint*, arXiv:2412.19437.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: Program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799. PMLR.

Yubin Ge, Salvatore Romeo, Jason Cai, Raphael Shu, Yassine Benajiba, Monica Sunkara, and Yi Zhang. 2025. TReMu: Towards neuro-symbolic temporal reasoning for LLM-agents with memory in multi-session dialogues. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 18974–18988, Vienna, Austria. Association for Computational Linguistics.

Zimu Lu, Aojun Zhou, Ke Wang, Houxing Ren, Weikang Shi, Junting Pan, Mingjie Zhan, and Hongsheng Li. 2025. Mathcoder2: Better math reasoning from continued pretraining on model-translated mathematical code. In *The Thirteenth International Conference on Learning Representations*.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, and 262 others. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. 2023. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 3806–3824.

Bhawna Piryani, Abdelrahman Abdullah, Jamshid Mozafari, Avishek Anand, and Adam Jatowt. 2025. It's high time: A survey of temporal information retrieval and question answering. *arXiv preprint arXiv:2505.20243*.

Xin Quan, Marco Valentino, Danilo Carvalho, Dhairya Dalal, and André Freitas. 2025. Peirce: Unifying material and formal reasoning via llm-driven neuro-symbolic refinement. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 11–21.

Dan Schumacher, Fatemeh Haji, Tara Grey, Niharika Bandlamudi, Nupoor Karnik, Gagana Uday Kumar, Jason Cho-Yu Chiang, Paul Rad, Nishant Vishwamitra, and Anthony Rios. 2025. Raster: Robust, agentic, and structured temporal reasoning. *Preprint*, arXiv:2406.19538.

Xintong Song, Bin Liang, Yang Sun, Chenhua Zhang, Bingbing Wang, and Ruifeng Xu. 2025. Bridging time gaps: Temporal logic relations for enhancing temporal reasoning in large language models. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 3040–3044.

Zhaochen Su, Juntao Li, Jun Zhang, Tong Zhu, Xiaoye Qu, Pan Zhou, Yan Bowen, Yu Cheng, and Min Zhang. 2024. Living in the moment: Can large language models grasp co-temporal reasoning? In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13014–13033, Bangkok, Thailand. Association for Computational Linguistics.

Md Nayem Uddin, Amir Saeidi, Divij Handa, Agastya Seth, Tran Cao Son, Eduardo Blanco, Steven Corman, and Chitta Baral. 2025. Unseentimeqa: Time-sensitive question-answering beyond llms' memorization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1873–1913.

Jiaying Wu and Bryan Hooi. 2025. Chain-of-timeline: Enhancing llm zero-shot temporal reasoning with sql-style timeline formalization. In *Workshop on Reasoning and Planning for Large Language Models*.

Siheng Xiong, Ali Payani, Ramana Kompella, and Faramarz Fekri. 2024. Large language models can learn temporal reasoning. *arXiv preprint arXiv:2401.06853*.

Ran Xu, Yuchen Zhuang, Yishan Zhong, Yue Yu, Xiangru Tang, Hang Wu, May Dongmei Wang, Peifeng Ruan, Donghan Yang, Tao Wang, and 1 others. 2025. Medagentgym: Training llm agents for code-based medical reasoning at scale. In *The Second Workshop on GenAI for Health: Potential, Trust, and Policy Compliance*.

A Dataset details

Within the Hard setting, there are two subsets: Serial and Parallel. In the Serial subset, all events occur sequentially. In the Parallel subset, events may occur concurrently, as permitted by the predefined scheduling rules. Questions in both subsets are further divided into three types: (1) Static-Time: The question refers to a fixed, absolute time point (e.g., "*Where is the package p1 at 12:23 AM*?"); (2) Relative-Time: The question refers to a time point relative to another event (e.g., "*Where is the package p3 5 hours before 03:41 PM*?"); (3) Hypothetical-Time: The question involves a hypothetical change to the schedule (e.g., "*If flying airplane a0 from location l2_0 to location l1_0 is delayed by 78 minutes, where is the package p2 at 09:02 PM*?"). The dataset statistics are summarized in Table 5.

Dataset	Q-Type	Number of Samples	Number of Events (Avg)
Serial	Static	900	26.93
	Relative	900	26.72
	Hypothetica	900	26.56
Parallel	Static	900	26.41
	Relative	900	26.38
	Hypothetica	900	26.51

Table 5: Statistic of UnseenTimeQA dataset under the Hard setting.

B Prompts for the Parser

We show our prompts for parser agent in Figure 7 and Figure 8

C Prompts for the Code generator and Debugger

We show our prompts for coder generator in Figure 9. The prompts utilized in the debugger module are presented in Figure 10

D Prompts for the Answerer

The prompts utilized in the answerer module are presented in Figure 11 and 12

Rewrite natural language to structured tuples: -The load event is rewritten as loading, package_id, vehicle_id, location, duration. -The unload event is rewritten as unloading, package_id, vehicle_id, location, duration. -The transport event is rewritten as transport, vehicle_id, from_location, to_location, duration. Example input: loading package p2 into truck t1 at location l1_1 takes 60 minutes to finish. Example output: loading,p2,t1,l1_1,60. Note: just output the tuples no more other text! Input: {Event}

Figure 7: Prompt for events parser

Parse the input question and output the reformatted version. Rules:
-New Question: Keep the first event and the final query ("where is..."), removing the "and" and the second event.
-Condition: Reformulate the second event (after "and") as:
-For load/unload: loading/unloading,expedite/delay,package_id,vehicle_id,location,time
-For transport: transport,expedite/delay,vehicle_id,from_location,to_location,time
Output Format: new question,event_type,expedite/delay,id1,id2/from_location,to_location/location(s),time

Figure 8: Prompt for question parser

You are an expert Python developer. Write a function schedule_events that schedules transportation/logistics events (loading, unloading, transport) while respecting dependencies and constraints. Output format: <think></think>
```python ...```
Rules:
-Loading or unloading a package in a vehicle is possible if the package and the vehicle are in the same location.-Packages can be loaded onto or unloaded from a same vehicle simultaneously, but loading and unloading cannot occur at the same time. -If any event is delayed or expedited, all subsequent dependent events are also delayed or expedited accordingly.-The transfer of the vehicle can only begin after all loading and unloading operations have been completed. According to the given rules, combined with events, conditions, and initial states, write a Python function to mark the start time and end time (minutes start at 0) for each event. Note: First,the python code should modify the duration of the corresponding event based on the given conditions (delay adds time, expedite subtracts time). Then infer the start and end times of each event based on the given rules.,
Example input: initial_states_description:['airplane a1 is at the location l1_0.',…], events_format:[('loading', 'p2', 't1', 'l1_1', 25), …], events_str:['loading package p2 into truck t1 at location l1_1 takes 25 minutes to finish.', …], condition:('transport', 'expedite', 't1', 'l1_1', 'l1_0', 25),
Example output:[(0, 'loading package p2 into truck t1 at location l1_1 takes 25 minutes to finish.', 25), …],

Figure 9: Prompt for coder generator

You are a debugger, providing you with a function output and a correct example (Ground truth output).-Based on the differences between the output(now output) and the example, determine whether the function has not yet met a certain rule, resulting in an error, and provide a modified version that complies with the rule. Output format: <think></think>
```python correct code ```, The function is used to add start and end time points (in minutes starting from 0) for each event based on the initial state, conditions(Possible values: transport, loading, unloading), and event list. Rules: {Serial rules or parallel rules}. python code: {Current Code} Input: {Current Code} Output: {Current Output} Ground truth output: {timeline}

Figure 10: Prompt for debugger

Given a question with AM/PM time and a timeline with virtual time starting from 0 min.-First, the time point of the question is always after the time point of the condition. Determine whether the statement of the question spans across days.-Second, Align the AM/PM time in the question with the virtual time on the timeline, and convert the AM/PM time into virtual time.-Finally, output the converted question.
Output format: Place the thinking steps in <think></think>, and place the converted questions in <q></q>

Figure 11: Prompt for absolute time conversion

Answer the question by reasoning step by step based on the information in the given timeline. The format of the timeline is (start_time,event,end_time).-When the package is in transit, the location is the name of the vehicle; -When the package is in the process of loading/unloading, the location is the name of the vehicle and the name of the location;-The location is the name of the location when the package is unloaded or before loading-Output format:<think>think steps</think><answer>final answer</answer>Timeline:(0, 'at location l1_2, package p0 is loaded into truck t1 and it takes 49 minutes to finish.', 49)…, question: converted questions

Figure 12: Prompt for answer question