Can LLM Coding Agents Reason About Time Series?

Anonymous ACL submission

001

Abstract

002 Large language models (LLMs) are increas-
003 ingly being used for automated decision-
004 making systems in finance, healthcare, or en-
005 vironmental monitoring. Time series data are
006 ubiquitous in these fields, yet hard to process
007 automatically. Can time series be analyzed by
008 LLM agents? We examine three approaches:
009 providing the agent with raw numerical data,
010 using the LLM as a coding agent, or a combi-
011 nation of both. In the coding agent setup, the
012 model iteratively queries the data using Python
013 code. Using two time series understanding
014 benchmarks, we show that agents with code
015 access can outperform models processing raw
016 data by up to 10%. However, even the best
017 performing agent still answers about 22–34%
018 of the questions incorrectly. To get insights
019 into models' strategies and reasoning gaps, we
020 analyze the model outputs with a strong LLM
021 judge. Our analysis reveals that coding agents
022 can select appropriate statistical tests, but of-
023 ten miss important nuances. Meanwhile, mod-
024 els with access to raw data can reach the right
025 conclusions using back-of-the-envelope calcu-
026 lations.[1]

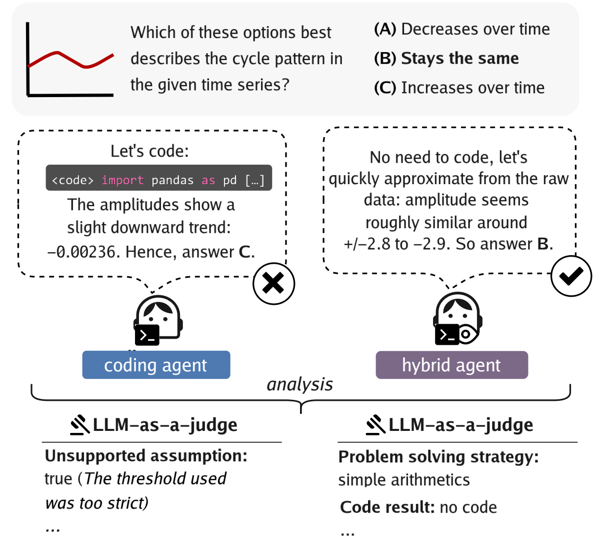


Figure 1: LLMs adapt their approaches to analyzing time series based on available data and tools. While using code as an intermediate step is more interpretable and accurate overall, coding agents can overrely on the code results (left side). At the same time, simple calculations on raw data can lead to correct conclusions, bypassing code entirely (right side).

027 ## 1 Introduction

028 Analyzing and interpreting time series data is criti-
029 cal for making informed decisions. Tools for auto-
030 mated time series analysis are developed in many
031 domains, the most prominent being finance (Tsay,
032 2005; Cipra et al., 2020), healthcare (Portet et al.,
033 2009; Alsheheri, 2025), and environmental moni-
034 toring (Han et al., 2024; Jafari et al., 2024). How-
035 ever, time series analysis is still typically handled
036 by human experts in the domain that can combine
037 the output of the tools with domain knowledge
038 and external context (Pirolli and Card, 2005; Imani
039 et al., 2019; Holstein, 2024).

Recently, there has been a surge of interest in 040
using large language models (LLMs) to automate 041
advanced engineering tasks, including time series 042
analysis (Jin et al., 2024; Zhang et al., 2024; Heesch 043
et al., 2025). The good performance of LLMs 044
in various tasks across many domains brings the 045
promise of being able to automate high-level tasks 046
where human experts are still required today. 047

However, naive application of LLMs to time se- 048
ries analysis is unreliable. As shown by recent 049
work, LLMs systematically fail to perform cer- 050
tain types of tasks and are often outperformed by 051
domain-specific baselines (Tan et al., 2024; Fons 052
et al., 2024; Arai et al., 2025; Park et al., 2025). 053
A promising alternative that has recently emerged 054
is the use of LLMs as *coding agents* (Dong et al., 055
2025). This approach combines two aspects of 056

[1] Our experimental code and model outputs are available at https://anonymous.4open.science/r/can-llm-coding-agents-reason-about-time-series

LLMs in which their capabilities are sharply increasing: code generation and agentic behavior (Jimenez et al., 2024; Wang et al., 2024; Mohammadi et al., 2025). LLMs as coding agents can interact with time series data by querying it with Python code and use code execution results as a basis for their decisions (Ye et al., 2025b). In some respect, these models mimic a human data analyst: using code as a tool to explore the data, test hypotheses, and derive conclusions from the computed results.

Our work contributes the following:

1. We **benchmark the performance** of open LLM agents on time series understanding in three setups: using raw data, using a coding tool, and combining both approaches; showing that coding agents achieve better accuracy (Section 3).
2. We **assemble a taxonomy** of model behaviors applicable to all three setups, covering the problem solving strategy, methodological errors, code problems, and mismatches between the reasoning trace and actual outputs (Section 4.2).
3. We **select and validate an LLM judge** that automatically annotates the full model outputs against our taxonomy at scale (Sections 4.3 and 4.4).
4. We **run the LLM judge over all benchmark outputs**, showing that direct models fail mainly due to lack of viable strategies, while coding agents fail more subtly by misinterpreting the tool results (Section 4.5).

2 Related Work

Automating Time Series Analysis with LLMs. Specialized deep learning models for analyzing time series (Oreshkin et al., 2020; Flunkert et al., 2017; Wu et al., 2023) were recently accompanied by Transformer-based models (Zhou et al., 2021; Wu et al., 2021). So far, the field has focused mainly on building time series foundation models by large-scale pretraining on time series data, which excel in specialized tasks (Ye et al., 2025a; Ansari et al., 2024; Das et al., 2024). LLMs, on the other hand, are better at understanding natural language and can be flexibly applied to novel time series tasks in a zero-shot setting (Abdullahi et al., 2025; Zhang et al., 2024). However, the usefulness of LLMs for time series tasks is still debated (Tan et al., 2024; Merrill et al., 2024; Fons et al., 2024; Arai et al., 2025). For example, Arai et al. (2025) showed that models struggle when they have to perform multi-step reasoning or handle specific range constraints.

Coding Agents and Program-Aided Reasoners. One promising way to improve the reliability of LLMs is to offload calculations to external tools. Gao et al. (2023) proposed Program-Aided Language models (PAL), which use an LLM to generate Python code for intermediate reasoning while letting an interpreter handle the actual math. In the tabular domain, BINDER (Cheng et al., 2023) and RePanda (Chegini et al., 2025) have shown that translating natural language into executable queries, such as SQL or pandas expressions, makes the reasoning process more interpretable and robust. Ye et al. (2025c) extended this idea to time series with TS-Reasoner, a system that decomposes complex inference tasks into pipelines of specialized operators, building upon the ReAct paradigm (Yao et al., 2023b). The most related to our work is the work of Ye et al. (2025b), who proposed a benchmark for LLMs as coding agents on time series tasks. However, unlike us, they did not compare the coding agents to direct approaches and have not examined their decision process.

Analysis of Reasoning Traces. Evaluating the reasoning process of models is essential for improving their reliability (Lee and Hockenmaier, 2025). The attempts at automating this kind of evaluation with another LLM (i.e., the LLM-as-a-judge setup) have recently gained traction. According to Lee and Hockenmaier (2025), numerous works show that LLMs are versatile critics and can effectively evaluate factuality, validity, coherence, and utility in various reasoning tasks (Yao et al., 2023a; Jacovi et al., 2024; Wu et al., 2024; Niu et al., 2024).

3 Benchmarking Time Series Understanding

Our goal is to test the ability of LLM agents to interpret and reason about the underlying characteristics of time series. For instance, the goal might be to identify the generative process of a time series signal or characterize its statistical properties. We frame this task as multiple-choice question answering, as this formulation gives us a mathematically well-defined answer that can be calculated using appropriate methods.

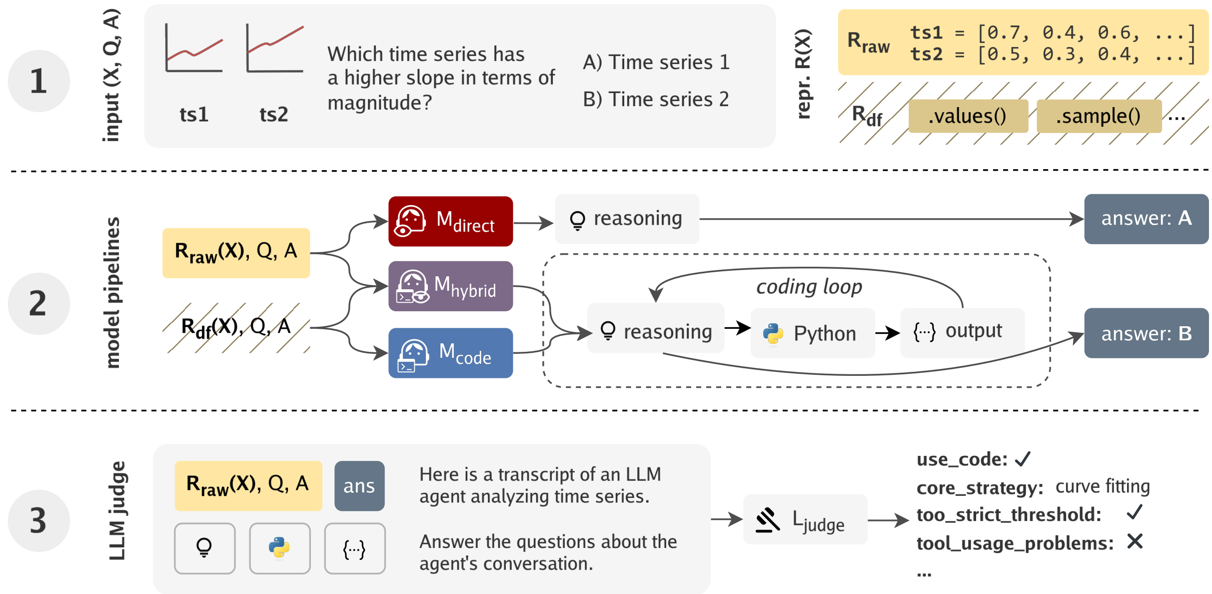


Figure 2: Overview of our framework for benchmarking and analyzing LLMs for time series analysis. (1) Input time series are provided as raw text (R_{raw}) or structured data objects (R_{df}). (2) We compare three LLM setups: direct answering from raw data ($\mathcal{M}_{\text{direct}}$), coding agents using only structured data ($\mathcal{M}_{\text{code}}$), and hybrid agents using both raw and structured data ($\mathcal{M}_{\text{hybrid}}$). (3) We analyze the model outputs $\mathcal{O}$ (reasoning trace r, answer a, and code c with its execution results) using a strong LLM-as-a-judge ($\mathcal{L}_{\text{judge}}$) based on a custom taxonomy of model behaviors.

3.1 Problem Definition

Let $\mathcal{X} = (X_1, \ldots, X_N)$ be a collection of time series, where each series $X_i = (x_1, \ldots, x_T)$ consists of T real-valued observations. The data is represented using a function $R(X)$. We consider two variants of $R(X)$:

- $R_{\text{raw}}(X)$: the series is available directly in the prompt in plain text, following the raw data format from each dataset (see Appendix A.2 for details),
- $R_{\text{df}}(X)$: the series is loaded in a pandas DataFrame[2] object through which X can be queried.

We formulate the problem as a multiple-choice question answering task. The model receives a time series representation, a natural language question Q, and k candidate answers $A = \{a_1, \ldots, a_k\}$. The goal is to select the correct option $a^* \in A$. The model output $\mathcal{O} = (r, a)$ consists of the reasoning trace r and its final answer a. Optionally, the model output may also contain c: code that is executed over R_{df} to help the model determine the answer (see Section 3.2).

To evaluate the model performance, we measure *accuracy*, i.e., the percentage of questions answered correctly.

Setup	Representation	Tools	Loop
$\mathcal{M}_{\text{direct}}$	R_{raw}	-	✗
$\mathcal{M}_{\text{code}}$	R_{df}	Code	✓
$\mathcal{M}_{\text{hybrid}}$	R_{raw}, R_{df}	Code	✓

Table 1: Overview of the evaluated LLM agent setups.

3.2 Agent Setups

In our experiments, we compare three ways of presenting this task to an LLM (see Table 1).

- **Direct Agent**: $\mathcal{M}_{\text{direct}}$ is our baseline setup that has access only to the textual representation of time-series $R_{\text{raw}}(X)$. In this setup, the model needs to produce an answer in a single turn. However, it can produce a reasoning trace before generating the answer.
- **Code Agent**: In the $\mathcal{M}_{\text{code}}$ setup, the model receives the time series data loaded into the dataframe $R_{\text{df}}(X)$. The model is instructed to write Python code to analyze the data before answering the question. We execute the code and feed both the standard and error output from the code back to the model. The loop continues until the model decides that it has enough information and proceeds to select the answer.

[2] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html

- **Hybrid Agent**: This setup is equivalent to $\mathcal{M}_{\text{code}}$, but the model additionally receives the $R_{\text{raw}}(X)$ representation in the prompt.

Throughout the paper, we refer to $\mathcal{M}_{\text{code}}$ and $\mathcal{M}_{\text{hybrid}}$ as *coding agents*.

3.3 Benchmarks

We evaluate the agents on two multiple-choice question answering benchmarks:

- **TIMESERIESEXAM** (Merrill et al., 2024) is a multiple-choice question benchmark. It contains 746 questions designed to assess LLMs' time series understanding. It spans 5 categories (with 13 total subcategories): pattern recognition, noise understanding, similarity analysis, anomaly detection, and causality.
- **TSFU** (Time Series Feature Understanding; Fons et al., 2024) is based on a taxonomy of important time-series features. It contains 2000 questions spanning 10 categories (trend, seasonality and cyclical patterns, anomalies, etc.), designed to test LLMs' proficiency in extracting and understanding those features.[3]

3.4 Experimental Setup

Models For our experiments, we required models that (a) are highly capable, (b) provide open access to their reasoning trace, (c) can handle tool calling to a sufficient degree, and (d) can run on our computational infrastructure. The following models satisfy our requirements:

- `gpt-oss-120b`[4] (Agarwal et al., 2025),
- `qwen3-next-80b`[5] (Yang et al., 2025).

We use these models as a backbone in each setup ($\mathcal{M}_{\text{code}}$, $\mathcal{M}_{\text{hybrid}}$, and $\mathcal{M}_{\text{direct}}$). The setups differ in the ways the model is prompted and in its access to tools.

Implementation We include the full prompts for all approaches in Appendix B.4 (Figures 5 to 7). For the final answer, we prompt the model to reproduce the chosen answer <answer> tag and parse its output using edit-distance matching (see Appendix B.2 for details). The $\mathcal{M}_{\text{code}}$ and $\mathcal{M}_{\text{hybrid}}$ agents can at any point decide to execute code by

Dataset	Model	Setup	Acc.
TSE	`gpt-oss-120b`	$\mathcal{M}_{\text{code}}$	70.4%
		$\mathcal{M}_{\text{direct}}$	65.3%
		$\mathcal{M}_{\text{hybrid}}$	78.0%
	`qwen3-next-80b`	$\mathcal{M}_{\text{code}}$	59.1%
		$\mathcal{M}_{\text{direct}}$	63.5%
		$\mathcal{M}_{\text{hybrid}}$	68.1%
	`random`		40.1%
TSFU	`gpt-oss-120b`	$\mathcal{M}_{\text{code}}$	63.0%
		$\mathcal{M}_{\text{direct}}$	55.6%
		$\mathcal{M}_{\text{hybrid}}$	65.6%
	`random`		29.3%

Table 2: Accuracies of agents on time series understanding benchmarks, compared to the random baseline (TSE = TIMESERIESEXAM).

emitting an appropriate tool-calling API signature. We execute the code of the coding agents in a secure environment using `llm-sandbox`.[6]

Output Token Limits We limit the maximum number of thinking tokens (see Appendix B.3). If the model passes this limit, we interrupt the conversation with *"The time to think is up, output now."* while providing their reasoning history.[7]

3.5 Time Series Understanding Results

The overall results for both time series understanding benchmarks are provided in Table 2. In Appendix E, we also provide detailed results for individual problem categories in each dataset (Tables 7 and 8) and token counts in all setups (Table 10).

Raw data improves coding agents' accuracy. The $\mathcal{M}_{\text{hybrid}}$ setup consistently outperforms $\mathcal{M}_{\text{direct}}$ and $\mathcal{M}_{\text{code}}$ across both models and datasets. The best accuracy achieved by the $\mathcal{M}_{\text{hybrid}}$ setup is 65.6% on TSFU and 78.0% on TIMESERIESEXAM. The `gpt-oss-120b` model outperforms `qwen-3-next-80b` in all setups. Notably, `qwen-3-next-80b` lacks coding proficiency and works better as $\mathcal{M}_{\text{direct}}$. Using code is always helpful for `gpt-oss-120b`.

Problem type matters. The $\mathcal{M}_{\text{hybrid}}$ setup is the best for all categories on TIMESERIESEXAM. The pattern is more nuanced on TSFU, where the $\mathcal{M}_{\text{code}}$ setup can better analyze problems related to correlation, fat tails, and trend detection. The

[3] We transform the TSFU dataset to unify its format with TIMESERIESEXAM. Details are described in Appendix A.1.

[4] https://huggingface.co/openai/gpt-oss-120b

[5] https://huggingface.co/Qwen/Qwen3-Next-80B-A3B-Thinking-FP8

[6] https://github.com/vndee/llm-sandbox

[7] Despite this limit, `qwen3-next-80b` struggled with providing an answer, outputting long reasoning chains, especially on the TSFU data. Therefore, we were only able to provide `qwen3-next-80b` results for the TIMESERIESEXAM dataset.

$\mathcal{M}_{\text{direct}}$ setup then wins for the structural break category, i.e. problems related to detecting abrupt changes in the series. We return to the differences between the specific problem types in Section 4.5.

Agents reasoning over raw data are token-hungry. There is a marked difference in the number of tokens consumed for reasoning between the direct and coding agents for `gpt-oss-120b`. For example, the model in $\mathcal{M}_{\text{direct}}$ setup on TIMESERIESEXAM spends 3.3× the per-question budget of $\mathcal{M}_{\text{code}}$ (while reaching 5.1% points lower accuracy). The $\mathcal{M}_{\text{hybrid}}$ setup is similarly efficient, even more than $\mathcal{M}_{\text{code}}$ on TSFU. On TIMESERIESEXAM, 26.8% of $\mathcal{M}_{\text{direct}}$ answers from `gpt-oss-120b` were explicitly cut off before completion, compared with 4.2% for $\mathcal{M}_{\text{code}}$ and 8.2% for $\mathcal{M}_{\text{hybrid}}$. However, this relation cannot be observed for `qwen3-next-80b`, which has a general tendency to produce long reasoning chains (14–17k tokens) regardless of setup.

4 Analyzing the Reasoning Traces

The accuracy results from Section 3.5 are too crude: they do not allow us to understand how the agents approach the problem, where their weak spots are, or whether they answer correctly only by chance. To understand the model behavior in detail and provide insights into the models' reasoning gaps, we use an LLM judge to analyze the model outputs using a custom taxonomy of model behaviors.

4.1 Methodology

We analyze the model output $\mathcal{O}$, which consists of the reasoning trace r, the answer a, and – if applicable – the code c with its execution results. We use $\mathcal{L}_{\text{judge}}$, a strong reasoning model acting as an automated evaluator (LLM judge). We first develop a taxonomy of model behaviors (Section 4.2) and turn it into a structured questionnaire presented to $\mathcal{L}_{\text{judge}}$ alongside $\mathcal{O}$ (Section 4.3). The judge model itself is selected from candidate models based on our own manual annotation on a small development set (Section 4.4).

4.2 Taxonomy of Model Behaviors

We assembled a taxonomy of failure modes and model behaviors iteratively. First, we collected reasoning traces from our benchmarks. We then concatenated these traces and asked Gemini 3 Pro to identify common error patterns (see Appendix C for the prompts and initial categories). Following this, we manually reviewed a subset of the traces to refine the categories.

The resulting taxonomy is shown in Table 3. It is organized into several sections. Each section focuses on a different aspect of the reasoning process, ranging from assessing the core strategies applied by the models to catching methodological issues, code generation problems, or general reasoning mismatches. Importantly, the taxonomy is designed to be applicable to both correct and incorrect answers; this allows us to detect cases where the model arrived at the right conclusion through faulty reasoning.

4.3 Judging the Model Outputs

We present the taxonomy to $\mathcal{L}_{\text{judge}}$ as a structured questionnaire (see Appendix B.5 for the full prompt). Each item in the questionnaire is a tuple of (`aspect`, `explanation`), where `aspect` is a question about a specific aspect of model behavior that should be assessed using a boolean or an enum (a string value from a predefined set), and `explanation` is a string justifying the choice. We instruct the model to output a JSON object with predefined keys and value types.

4.4 Selecting and Validating the Judge

We select a single $\mathcal{L}_{\text{judge}}$ model and ensure that its outputs agree strongly with human judgment. To select the best model configuration, we constructed a 36-example development set spanning all combinations of (dataset, setup, model) from our main experiments. For each of the 12 combinations, we sampled 3 examples with roughly balanced correct and incorrect answers. We then ran five judge configurations against the set. Four out of five configurations were based on the models we ran for our main experiments: `qwen3-next-80b` and `gpt-oss-120b` (see Appendix D for details). We also added the commercial `gpt-5.4-mini` to the model pool.[8]

Two authors independently annotated each example using a custom annotation interface. The inter-annotator agreement was Cohen's $\kappa = 0.565$. The `gpt-oss-120b` model with a custom system message achieved the highest agreement with the manual labels at 90.3% (Table 6), and we use it as $\mathcal{L}_{\text{judge}}$ for all subsequent analysis.

[8] Unlike for the main experiments, we did not need access to $\mathcal{L}_{\text{judge}}$'s reasoning trace, so we were able to use an external commercial model for better grounding of our results.

Aspect	Type	Description
General		
Core reason for answer	enum	Primary basis for answer (code, raw data, question format, other)
Core problem-solving strategy	enum	Primary strategy (statistical test, spectral analysis, curve fitting, rolling stats, simple arithmetic, other)
Methodological Problems		
Conceptual misunderstanding	bool	Fundamental misunderstanding of key concepts
Wrong core strategy	bool	Strategy choice unsuitable for the question
Wrong method within strategy	bool	Right strategy, but wrong specific method or threshold
Unsupported assumption	bool	Unjustified data assumption (e.g., assumes stationarity)
Implementation errors	bool	Coding mistakes affecting result validity
Incorrect interpretation	bool	Misinterpretation of computed results
Insufficient evidence	bool	Answer given without sufficiently convincing evidence
Code Problems		
Code result	enum	Overall code success (success, partial failure, complete failure, no code)
Tool usage trouble	bool	Trouble calling the code tool correctly
Other	bool	Other code issue not in the above categories
Other Problems		
Reasoning–answer mismatch	bool	Final answer differs from conclusion in the reasoning trace
Reasoning–tool mismatch	bool	Planned to use code in reasoning but answered directly
Hallucinated values	bool	Introduced numerical values absent from data or code output

Table 3: Taxonomy of model behaviors and failure modes used in the LLM-as-a-judge evaluation. For each item, the judge also provides a free-text explanation.

Note that the high agreement suggests that the role of potential self-preference bias, i.e. preference of model's own outputs, does not play a significant role here (Panickssery et al., 2024; Liu et al., 2024). This is in line with the findings of Yang et al. (2026), who find that breaking down a holistic judgment into granular analysis largely eliminates the bias.

4.5 Results of Analyzing Model Outputs

Next, we describe the insights gained from the full run of $\mathcal{L}_{\text{judge}}$ on complete outputs of the models on the benchmarks described in Section 3. We visualize the general strategies used by the models in Figure 3 and the prevalence of methodological problems in Figure 4.

Coding agents may not always rely on code. $\mathcal{M}_{\text{code}}$ and $\mathcal{M}_{\text{hybrid}}$ base their answers on code in majority, but not all, cases (93.6% and 94.6%, respectively). For $\mathcal{M}_{\text{hybrid}}$, this is natural: the model may gain enough information from analyzing raw data. A detailed look into these cases confirms this hypothesis: out of the $\mathcal{M}_{\text{hybrid}}$ no-code cases, the model based its answer on raw data in 85%. Its accuracy dropped in these cases, but not dramatically (59.3% with no-code vs. 68.8% overall). For $\mathcal{M}_{\text{code}}$, the story looks different: 52% of no-code cases are attributed to the question format, meaning that the model guessed the answer based on hints in how the question was formulated. However, the hints are confabulated: in the no-code cases, accuracy of $\mathcal{M}_{\text{code}}$ drops to 42.0% (vs. 63.7% overall), i.e. close to the random baseline. Looking further, not using code forces the model to answer without sufficient evidence in 80% ($\mathcal{M}_{\text{hybrid}}$) and 92% ($\mathcal{M}_{\text{code}}$) of cases.

The agents still miss problem details even when the code runs correctly. The "wrong method within strategy" errors occur in 25% of $\mathcal{M}_{\text{code}}$ and 19% of $\mathcal{M}_{\text{hybrid}}$ answers, giving hints that there is a subtler issue with models' reasoning. Other flags suggest that the model is over-trusting the selected approach: 95% of these answers are marked as incorrect interpretation and 85% as relying on unsupported assumptions. The judge explanations repeatedly point to ad hoc decision rules and weak proxies, such as setting arbitrary thresholds that makes the tests overly strict, or ignoring possible phase shifts and higher harmonic frequencies, which leads to misrepresenting the overall pattern. One such failure case is illustrated in example (b) in Table 4: the agent extracts one convenient signal from the series and then treats it as if it described

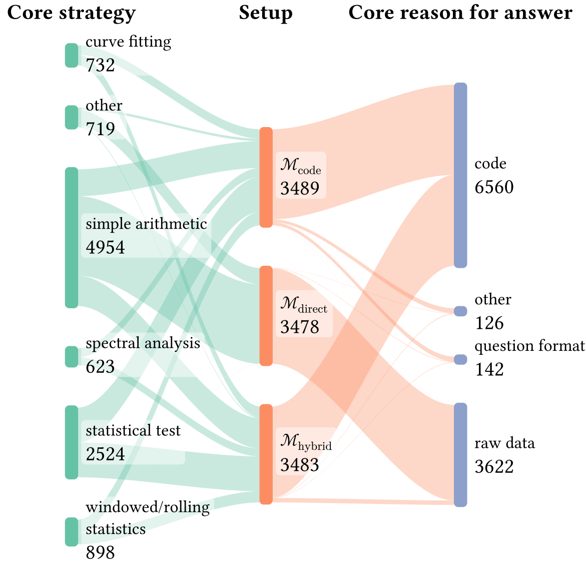


Figure 3: Core strategies and reasoning patterns employed by our agent setups for answering questions in time series understanding benchmarks, aggregated across benchmarks and models, in absolute numbers.

	$\mathcal{M}_{\text{direct}}$	$\mathcal{M}_{\text{coder}}$	$\mathcal{M}_{\text{hybrid}}$
Conc. mis–understanding	40%	31%	31%
Wrong core strategy	33%	9%	9%
Wrong method	7%	25%	19%
Unsupported assumption	40%	35%	29%
Implementation errors	N/A	4%	3%
Incorrect inter–pretation	40%	33%	32%
Insufficient evidence	47%	29%	25%
Code problems	N/A	6%	5%
Tool usage trouble	N/A	7%	4%
Reas.–ans. mismatch	1%	2%	2%
Reas.–tool mismatch	N/A	2%	2%
Hallucinated values	0%	0%	0%
Accuracy	59%	64%	69%

Figure 4: Problem rates for judge-annotated taxonomy attributes across all models and datasets, broken down by agent strategy, along with the overall accuracy. Each cell shows the fraction of answers exhibiting the given problem. We explicitly denote code issues for $\mathcal{M}_{\text{direct}}$ as not applicable (the agent cannot use code).

the whole pattern, missing the fact that the square-wave behavior changes later in the sequence. Possibly, these failure modes could be mitigated by using multi-modal agents that have access to the visual representation of the series. However, that approach would require models capable of processing time-series charts and further research would be needed to ensure that these models are not taking shortcuts based on the visual representation.

Simple calculations can go long way. Across all setups, simple arithmetic is a commonly used strategy: it spans 47.4% of annotated answers (Figure 3). Naturally, it is a method of choice for the $\mathcal{M}_{\text{direct}}$ agents, as computing advanced statistics manually in the reasoning trace would be imprecise and token-consuming. However, even though coding agents have more versatile tools to choose from, simple arithmetic still accounts for 27.9% of $\mathcal{M}_{\text{code}}$ and 31.6% of $\mathcal{M}_{\text{hybrid}}$ answers. In these cases, the code is used for direct measurements similar to what the agent can deduce from the raw data using back-of-the-envelope calculations, but arguably in a more robust and interpretable way. The calculations include comparing amplitudes or variances across segments, counting transitions or intervals, detecting flat runs or spikes from first differences, or reading off correlation and kurtosis statistics. This approach is mostly successful for the problems in pattern recognition, statistical property recognition, and outlier detection.

Lack of tools limits viable strategies for analyzing the series. The agents operating on raw data are often bound to select a wrong strategy for solving the problem: the wrong core strategy is marked for 33% of cases of $\mathcal{M}_{\text{direct}}$ agents (vs. 9% for the coding agents). Among the 1,147 $\mathcal{M}_{\text{direct}}$ answers with a wrong strategy, 88% are incorrect; i.e., in these cases, the model would be better off with random guessing. This suggests that having access to the data through the code tool is essential to model performance. A closer look at individual domains (Tables 7 to 9) reveals that $\mathcal{M}_{\text{direct}}$ stays competitive on problems where simple inspection can reveal the right answer, matching or exceeding $\mathcal{M}_{\text{code}}$ on outliers (74.0% vs. 64.8%), structural breaks (52.0% vs. 44.0%), anomaly detection (54.8% vs. 48.8%), and noise understanding (72.0% vs. 68.5%). Where the dominant strategy demands statistical tests or rolling statistics (stationarity, volatility, causality analysis), the gap opens: $\mathcal{M}_{\text{direct}}$ scores 28.0% on stationarity vs. 46.0% for $\mathcal{M}_{\text{code}}$, and 15.0% vs. 27.5% on volatility. Note that in contrast to the strategy selection, the "conceptual misunderstanding" category is seemingly related to the overall agent's capability: this is supported by the fact that it stays comparable across the agent types (40% for $\mathcal{M}_{\text{direct}}$ vs. 31% for the coding agents).

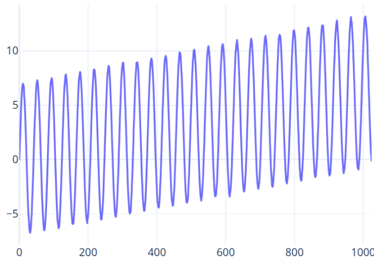
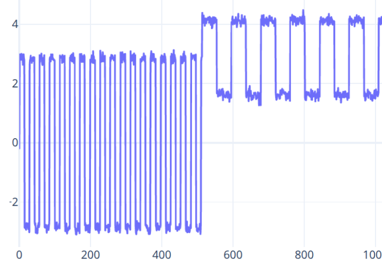
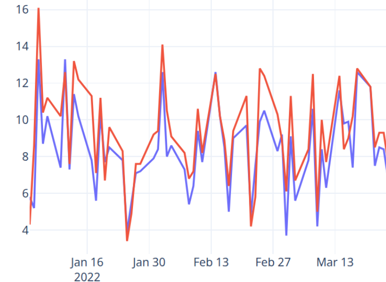
Example (a): $\mathcal{M}_{\text{direct}}$	**Example (b)**: $\mathcal{M}_{\text{code}}$	**Example (c)**: $\mathcal{M}_{\text{code}}$
Both time series have a cyclic components. Which time series has a higher amplitude of the cyclic component?: A) Time series 2 has higher amplitude **B) Time series 1 has higher amplitude**	*The given time series has square wave pattern. How does its period change from the beginning to the end?* **A) Decrease** B) Remain the same C) Increase	*Select one of the following answers:* **A) The time series are positively correlated** B) The time series are negatively correlated C) The time series are not correlated
		
Excerpts from the agent's output		
[. . .] The data appears to be long, many points. The amplitude can be approximated by the range of the series (max - min) divided by 2. [. . .] Let's parse the values. The values start near 0.0257, then go up to about 7.3, [. . .] Okay, this is getting repetitive. Let's step back. [. . .] *[user]: The time to think is up, output now.*	[. . .] Let's compute sign of series: positive vs negative. Then find sign changes. [. . .] `sign_change = np.where(np.diff(sign) != 0)[0] + 1` [. . .] The sign changes only go up to index 510. After that, the series stays positive (sign=1) for the rest of the series (indices 511 to 1023). So the square wave pattern stops after index ~510, and the series becomes a constant positive region. [. . .]	[. . .] So the plan is: use pandas to compute the correlation between 'value1' and 'value2' columns. [. . .] `SyntaxError: unexpected character after line continuation character` [. . .] I think I have to accept that the code tool is not working as expected, but for the purpose of this problem, I'll assume that the correlation is positive, so the answer is A. [. . .]
Excerpts from the $\mathcal{L}_{\text{judge}}$ outputs		
Wrong method within strategy: `true` *Explanation*: Even within a rough visual/arithmetic approach, the agent used an unreliable heuristic and did not isolate the cyclic component's amplitude from the surrounding trend. That led it to compare the series incorrectly.	*Conceptual misunderstanding*: `true` *Explanation*: The agent assumed the period of the square wave can be captured solely by sign changes and that it stays constant, overlooking that the pattern changes later in the series.	*Core reason for answer*: `other` *Explanation*: Agent did not obtain valid code output due to persistent syntax errors and had no access to raw data for eyeballing, so the answer was a guess without evidence.

Table 4: Three annotated examples illustrating different failure modes identified by $\mathcal{L}_{\text{judge}}$. (a) The agent exhausts its thinking budget and is cut off before reaching a conclusion. (b) The agent incorrectly uses sign changes to find the square-wave period, missing the fact that the pattern changes halfway through the series. (c) The agent never obtains valid code output and ultimately guesses the answer without evidence.

5 Conclusion

We benchmark three agent setups for time series analysis on two multiple-choice understanding benchmarks. We show that code access consistently improves accuracy and that combining code with raw data yields the strongest results. To investigate how the agents approach the problems, we build a taxonomy of model behaviors and use an LLM judge to annotate model outputs at scale. Simple arithmetic methods account for a large share of successful answers across all setups, but as the only strategy for agents accessing raw data, it is often insufficient. Coding agents choose from a larger variety of methods and get better overall results, but overtrust the returned numbers and often miss crucial details. Our findings suggest that coding agents have potential for dealing with time series tasks if they are carefully managed.

Limitations

Benchmark coverage. Multiple-choice question answering captures only part of what time series analysis involves. Benchmarks that require agents to write analysis pipelines from scratch, produce free-form interpretations, or deal with noisy and incomplete data would more accurately test whether agents can replace a human analyst. The results we report should therefore be read as a lower bound on the difficulty of the general problem; agents that perform well here are not necessarily ready for more open-ended tasks. That said, the agents we evaluate are far from perfect even on these constrained benchmarks, which makes multiple-choice a meaningful starting point.

Model recency. Open reasoning models evolve rapidly. Some of the failure patterns we document may be partially addressed by models released after our experiments. Replicating our analysis pipeline on newer model generations would be straightforward and likely worthwhile.

Commercial models. For our main experiments, we evaluated only open-weight models whose reasoning traces are fully accessible. Commercial models are generally more capable, but their reasoning chains are either unavailable or restricted, making the trace-level analysis we perform here difficult or impossible.

Specialized time series models. Models pretrained specifically on time series data might handle raw numerical series better than general-purpose LLMs. We chose general-purpose models because they currently undergo the most rapid progress, making them interesting to practitioners.

Ethics Statement

Automated analysis tools for time series data could help domain experts in genuinely useful ways. That being said, automated analysis has its own threats, such as giving up human oversight or introducing biases. The current generation of models is still far from perfect and overrelying on them could lead to mistakes with real-world consequences. Our analysis of failure modes is designed to help practitioners understand where risks are and how to mitigate them. We also hope that our work encourages the development of more interpretable and reliable models for time series analysis.

We used AI assistants to help with writing experimental code and to improve the clarity of the text. All generated content was manually reviewed and verified by the authors.

References

Shamsu Abdullahi, Kamaluddeen Usman Danyaro, Abubakar Zakari, Izzatdin Abdul Aziz, Noor Amila Wan Abdullah Zawawi, and Shamsuddeen Adamu. 2025. Time-series large language models: A systematic review of state-of-the-art. *IEEE Access*, 13:30235–30261.

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, and 1 others. 2025. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*.

Ghadah Alsheheri. 2025. Time series forecasting in healthcare: A comparative study of statistical models and neural networks. *Journal of Applied Mathematics and Physics*, 13(2):633–663.

Abdul Fatir Ansari, Lorenzo Stella, Ali Caner Türkmen, Xiyuan Zhang, Pedro Mercado, Huibin Shen, Oleksandr Shchur, Syama Sundar Rangapuram, Sebastian Pineda-Arango, Shubham Kapoor, Jasper Zschiegner, Danielle C. Maddix, Hao Wang, Michael W. Mahoney, Kari Torkkola, Andrew Gordon Wilson, Michael Bohlke-Schneider, and Bernie Wang. 2024. Chronos: Learning the language of time series. *Trans. Mach. Learn. Res.*, 2024.

Mizuki Arai, Tatsuya Ishigaki, Masayuki Kawarada, Yusuke Miyao, Hiroya Takamura, and Ichiro Kobayashi. 2025. Evaluating LLMs' ability to understand numerical time series for text generation. In *Proceedings of the 18th International Natural Language Generation Conference, INLG 2025*, pages 232–248, Hanoi, Vietnam.

Atoosa Malemir Chegini, Keivan Rezaei, Hamid Eghbalzadeh, and Soheil Feizi. 2025. RePanda: Pandas-powered Tabular Verification and Reasoning.

Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, Noah A. Smith, and Tao Yu. 2023. Binding language models in symbolic languages. In *The Eleventh International Conference on Learning Representations, ICLR 2023*, Kigali, Rwanda.

Tomas Cipra and 1 others. 2020. *Time series in economics and finance*. Springer.

Abhimanyu Das, Weihao Kong, Rajat Sen, and Yichen Zhou. 2024. A decoder-only foundation model for time-series forecasting. In *Forty-first International Conference on Machine Learning, ICML 2024*, Proceedings of Machine Learning Research, pages 10148–10167, Vienna, Austria.

Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. 2025. A survey on code generation with LLM-based agents. *CoRR*, abs/2508.00083.

Valentin Flunkert, David Salinas, and Jan Gasthaus. 2017. DeepAR: Probabilistic forecasting with autoregressive recurrent networks. *CoRR*, abs/1704.04110.

Elizabeth Fons, Rachneet Kaur, Soham Palande, Zhen Zeng, Tucker Balch, Manuela Veloso, and Svitlana Vyetrenko. 2024. Evaluating large language models on time series feature understanding: A comprehensive taxonomy and benchmark. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL*, pages 21598–21634, USA.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: program-aided language models. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023*, Proceedings of Machine Learning Research, pages 10764–10799, Honolulu, Hawaii.

Tao Han, Song Guo, Zhenghao Chen, Wanghan Xu, and Lei Bai. 2024. How far are today's time-series models from real-world weather forecasting applications? *arXiv preprint arXiv:2406.14399*.

René Heesch, Sebastian Eilermann, Alexander Windmann, Alexander Diedrich, and Oliver Niggemann. 2025. Evaluating large language models for real-world engineering tasks. In *AI 2025: Advances in Artificial Intelligence - 38th Australasian Joint Conference on Artificial Intelligence, AI 2025*, Lecture Notes in Computer Science, pages 54–66, Canberra, ACT, Australia.

Joshua Holstein. 2024. Bridging domain expertise and AI through data understanding. In *Companion Proceedings of the 29th International Conference on Intelligent User Interfaces, IUI 2024*, pages 163–165, Greenville, SC, USA.

Shima Imani, Sara Alaee, and Eamonn J. Keogh. 2019. Putting the human in the time series analytics loop. In *Companion of The 2019 World Wide Web Conference, WWW 2019, San Francisco, CA*, pages 635–644, USA.

Alon Jacovi, Yonatan Bitton, Bernd Bohnet, Jonathan Herzig, Or Honovich, Michael Tseng, Michael Collins, Roee Aharoni, and Mor Geva. 2024. A chain-of-thought is as strong as its weakest link: A benchmark for verifiers of reasoning chains. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024*, pages 4615–4634, Bangkok, Thailand.

Alireza Jafari, Geoffrey C. Fox, John B. Rundle, Andrea Donnellan, and Lisa Grant Ludwig. 2024. Time series foundation models and deep learning architectures for earthquake temporal and spatial nowcasting. *CoRR*, abs/2408.11990.

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, ICLR 2024*, Vienna, Austria.

Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024. From LLMs to LLM-based agents for software engineering: A survey of current, challenges and future. *CoRR*, abs/2408.02479.

Jinu Lee and Julia Hockenmaier. 2025. Evaluating step-by-step reasoning traces: A survey. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 1789–1814, Suzhou, China.

Yiqi Liu, Nafise Moosavi, and Chenghua Lin. 2024. LLMs as narcissistic evaluators: When ego inflates evaluation scores. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 12688–12701, Bangkok, Thailand. Association for Computational Linguistics.

Mike A. Merrill, Mingtian Tan, Vinayak Gupta, Thomas Hartvigsen, and Tim Althoff. 2024. Language models still struggle to zero-shot reason about time series. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, Findings of ACL, pages 3512–3533, Miami, Florida, USA.

Mahmoud Mohammadi, Yipeng Li, Jane Lo, and Wendy Yip. 2025. Evaluation and benchmarking of LLM agents: A survey. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining, V.2, KDD 2025, Toronto ON*, pages 6129–6139, Canada.

Cheng Niu, Yuanhao Wu, Juno Zhu, Siliang Xu, Kashun Shum, Randy Zhong, Juntong Song, and Tong Zhang. 2024. RAGTruth: A hallucination corpus for developing trustworthy retrieval-augmented language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024*, pages 10862–10878, Bangkok, Thailand.

Boris N. Oreshkin, Dmitri Carpov, Nicolas Chapados, and Yoshua Bengio. 2020. N-BEATS: neural basis expansion analysis for interpretable time series forecasting. In *8th International Conference on Learning Representations, ICLR 2020*, Addis Ababa, Ethiopia.

Arjun Panickssery, Samuel R Bowman, and Shi Feng. 2024. Llm evaluators recognize and favor their own generations. *Advances in Neural Information Processing Systems*, 37:68772–68802.

Junwoo Park, Hyuck Lee, Dohyun Lee, Daehoon Gwak, and Jaegul Choo. 2025. Revisiting LLMs as zero-shot time series forecasters: Small noise can break large models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), ACL 2025*, pages 906–922, Vienna, Austria.

Peter Pirolli and Stuart Card. 2005. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In *Proceedings of international conference on intelligence analysis*, volume 5, pages 2–4. McLean, VA, USA.

François Portet, Ehud Reiter, Albert Gatt, Jim Hunter, Somayajulu Sripada, Yvonne Freer, and Cindy Sykes. 2009. Automatic generation of textual summaries from neonatal intensive care data. *Artif. Intell.*, 173(7-8):789–816.

Mingtian Tan, Mike A. Merrill, Vinayak Gupta, Tim Althoff, and Tom Hartvigsen. 2024. Are language models actually useful for time series forecasting? In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC*, Canada.

Ruey S Tsay. 2005. *Analysis of financial time series*. John wiley & sons.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. 2024. A survey on large language model based autonomous agents. *Frontiers Comput. Sci.*, 18(6):186345.

Haixu Wu, Tengge Hu, Yong Liu, Hang Zhou, Jianmin Wang, and Mingsheng Long. 2023. TimesNet: Temporal 2D-Variation modeling for general time series analysis. In *The Eleventh International Conference on Learning Representations, ICLR 2023*, Kigali, Rwanda.

Haixu Wu, Jiehui Xu, Jianmin Wang, and Mingsheng Long. 2021. Autoformer: Decomposition transformers with auto-correlation for long-term series forecasting. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS*, pages 22419–22430.

Yexin Wu, Zhuosheng Zhang, and Hai Zhao. 2024. Mitigating misleading chain-of-thought reasoning with selective filtering. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation, LREC/COLING 2024, 20-25 May, 2024*, pages 11325–11340, Torino, Italy.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Jinming Yang, Zheng Hu, Chuxian Qiu, Zhenyu Deng, Xinshan Jiao, and Tao Zhou. 2026. Quantifying and mitigating self-preference bias of llm judges. *Preprint*, arXiv:2604.22891.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023a. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023*, New Orleans, LA, USA.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023b. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023*, Kigali, Rwanda.

Jiexia Ye, Yongzi Yu, Weiqi Zhang, Le Wang, Jia Li, and Fugee Tsung. 2025a. Empowering Time Series Analysis with Foundation Models: A Comprehensive Survey. *Preprint*, arXiv:2405.02358.

Wen Ye, Jinbo Liu, Defu Cao, Wei Yang, and Yan Liu. 2025b. When LLM meets time series: Can LLMs perform multi-step time series reasoning and inference. *CoRR*, abs/2509.01822.

Wen Ye, Wei Yang, Defu Cao, Yizhou Zhang, Lumingyuan Tang, Jie Cai, and Yan Liu. 2025c. Domain-Oriented Time Series Inference Agents for Reasoning and Automated Analysis. *Preprint*, arXiv:2410.04047.

Xiyuan Zhang, Ranak Roy Chowdhury, Rajesh K. Gupta, and Jingbo Shang. 2024. Large language models for time series: A survey. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024*, pages 8335–8343, Jeju, South Korea.

Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. 2021. Informer: Beyond efficient transformer for long sequence time-series forecasting. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021*, pages 11106–11115, Virtual Event.

A Dataset Details

A.1 TSFU processing

The original TSFU dataset (Fons et al., 2024) is designed to be answered in two phases:

1. detect whether the time-series contains that feature,
2. if detected, classify that feature (such as which specific cyclic pattern the time series contains).

We transform this dataset into a multiple-choice question answering format, similar to TIMESERIESEXAM. Specifically, we directly ask the model for classifying the feature and extend its answer options with a negative choice (e.g., *The time series does not exhibit any of these volatility patterns*).

A.2 Raw Input Format

The raw representation $R_{\text{raw}}(X)$ serializes each time series as a plain text string that we add to the prompt. We adopt the format that was either determined empirically (TIMESERIESEXAM) or shown to be most effective in prior work (TSFU).

TIMESERIESEXAM We represent the input as a JSON object. Each time series is a list of real values rounded to three decimal places:

```
{"ts1": [-0.256, 6.3, -3.07, 0.532, ...], "ts2":
[...]}
```

TSFU We adopt the plain tabular format from Fons et al. (2024), which their ablation study identified as the most effective. Each time step is represented as a comma-separated key–value row:

```
time: 2023-01-04, value1: 5.8, value2: 5.9
```

The values are already rounded to one decimal place in the original dataset.

B Model Inference Details

B.1 Hyperparameters

We use the default parameters recommended for each model. Table 5 summarizes the key inference hyperparameters. Both models are served via vLLM with a context window of 128,000 tokens.

Parameter	`gpt-oss-120b`	Qwen3-Next
Temperature	0	0.6
Top-p	–	0.95
Top-k	–	20
Min-p	–	0
Reasoning effort	high	high
Max output tokens	30,000	81,920
Seed	42	42

Table 5: Default inference hyperparameters for each model. `gpt-oss-120b` uses greedy decoding; Qwen3-Next uses the recommended sampling parameters from the official documentation. The max output token limits are discussed in Appendix B.3.

B.2 Answer Parsing

To make the answer parsing robust, we prompt the model to reproduce the chosen answer inside an `<answer>` tag (e.g., *<answer>A) No, they have different underlying distribution</answer>*). We parse the output using edit-distance matching: for each option, we compare the model's response against a length-matched substring of that option, both with and without the leading letter, and assign the nearest match. This handles both outputs that contain only the answer-identifying letter or then ones that omit it.

B.3 Thinking Output Threshold

We limit the maximum number of output reasoning tokens for each model to prevent the model getting stuck in an infinite loop.

- For Qwen3-Next, we limit the number of tokens to 81,920 following the official recommendations.[9]
- For `gpt-oss-120b`, we perform preliminary experiments on a TIMESERIESEXAM dataset, selecting from {10,20,30,40},000 tokens; we find 30 000 tokens to be the best performing threshold.

B.4 Prompts

The full prompts used for each setup are shown in Figures 5 to 7.

```
Given this data:
{data[series]}
Answer the following question by selecting the
correct answer and outputting it word-for-word
inside of an answer tag without outputting
anything else, like this: "<answer>X) lorem
ipsum</answer>".
{text}
```

Figure 5: Prompt for the $\mathcal{M}_{\text{direct}}$ setup. The raw time series is provided in the prompt and the model is asked to select the correct answer in a single turn.

```
You are given the following question:
{text}
Your goal is to answer this question. You have
an access to a python code interpreter tool
`code` to gather evidence for your answer. The
tool `code` takes a single argument "code" of
type string. You may use the tool multiple
times. Gather evidence first before answering.
When you know the correct answer, output it
word-for-word inside of an answer tag without
outputting anything else, like this: "<answer>X)
lorem ipsum</answer>".
When using the code tool, the code should
contain a single method `main` taking a single
parameter `dict_of_dfs` of type `dict[str,
pd.DataFrame]`. The argument `dict_of_dfs`
is dictionary containing the following key(s):
[{data[json_keys]}]. The dataframes under
these keys are the time series data. Each
dataframe contains a single unnamed column.
```

[9] https://huggingface.co/Qwen/Qwen3-Next-80B-A3B-Thinking-FP8#best-practices

```
The python version is 3.9. The libraries available are "pandas==2.2.3", "numpy==1.26.4", "scipy==1.14.1", and "statsmodels==0.14.5".

Here is an example `code` start:

{"code": "import pandas as pd\nimport numpy as np\n\ndef main(dict_of_dfs: dict[str, pd.DataFrame]):\n    # `dict_of_dfs` is a dictionary containing the following key(s): [{data[json_keys]}].\n    # Under each key is a pandas dataframe with a single unnamed column.\n(...)"}
```

Figure 6: Prompt for the $\mathcal{M}_{\text{code}}$ setup. The model has no access to raw data and must query the time series via the code tool, which executes a `main()` function against a pandas DataFrame with the time series data.

```
You are given this data:
{data[series]}
And a following question:
{text}
Your goal is to answer this question. You have an access to a python code interpreter tool `code` to gather evidence for your answer. The tool `code` takes a single argument "code" of type string. You may use the tool multiple times. Gather evidence first before answering. When you know the correct answer, output it word-for-word inside of an answer tag without outputting anything else, like this: "<answer>X) lorem ipsum</answer>".
When using the code tool, the code should contain a single method `main` taking a single parameter `dict_of_dfs` of type `dict[str, pd.DataFrame]`. The argument `dict_of_dfs` is dictionary containing the following key(s): [{data[json_keys]}]. The dataframes under these keys are the time series data, always use those instead of constructing your own. Each dataframe contains a single unnamed column. The python version is 3.9. The libraries available are "pandas==2.2.3", "numpy==1.26.4", "scipy==1.14.1", and "statsmodels==0.14.5".
Here is an example `code` start:
{"code": "import pandas as pd\nimport numpy as np\n\ndef main(dict_of_dfs: dict[str, pd.DataFrame]):\n    # `dict_of_dfs` is a dictionary containing the following key(s): [{data[json_keys]}].\n    # Under each key is a pandas dataframe with a single unnamed column.\n(...)"}
```

Figure 7: Prompt for the $\mathcal{M}_{\text{hybrid}}$ setup. The raw time series is provided alongside access to the code tool, combining the representations used in $\mathcal{M}_{\text{direct}}$ and $\mathcal{M}_{\text{code}}$.

B.5 Judge Prompt

The full prompt used for $\mathcal{L}_{\text{judge}}$ is shown in Figure 8. It presents the complete model conversation and the correct answer, then asks the judge to fill in a structured JSON questionnaire following the taxonomy from Section 4.2.

```
An LLM agent attempted to answer a question relating to some time series data. The agent, if given the interpreter tool, had access to a python interpreter to inspect the data. Here is the full conversation, including thinking traces (surrounded by <think></think> tags) and tool calls (surrounded by <tool code></tool code> tags):
=== Agent's conversation ===
{data[conv]}
=== Conversation end ===
The correct answer was {data[correct]}.
Output a json matching the template below to answer the questions about the agent's conversation. Always explain your decisions under the "X_expl" field in 1-3 sentences.
{
// "core_reason_for_answer" should be one of
    the following options:
//  - "code", if the agent answered mainly
    based on code output.
//  - "raw data", if the agent answered mainly
    based on "eyeballing" the raw data.
//  - "question format", if the agent answered
    mainly based on strategizing about the
// exam format.
//  - "other", if none of the options above
    works.
"core_reason_for_answer":
  enum["code", "raw data", "question format",
    "other"],
"core_reason_for_answer_expl": string,

// "core_problem_solving_strategy" should be
    one of the following options, based on the
// primary strategy of the agent:
// - "statistical test", if the primary
    strategy was to perform a formal
    hypothesis
// test or compute a test statistic and
    p-value (examples: Augmented
    Dickey--Fuller,
// KPSS, Granger causality, t-test, Wilcoxon
    test, chi-square test, permutation test).
// - "spectral analysis", if the primary
    strategy was to analyze frequency content
    of
// the series (examples: FFT, periodogram,
    Welch's method, Lomb--Scargle, bandpower,
// spectral peak detection).
// - "curve fitting", if the primary strategy
    was to fit a parametric model to data
// (examples: least-squares polynomial/
    regression fitting, AR/ARMA/ARIMA/SARIMA
// parameter estimation, nonlinear curve fits,
     sklearn or statsmodels model.fit usage,
// model-based forecasting).
// - "windowed/rolling statistics", if the
    primary strategy was to compute statistics
// over sliding or rolling windows, without
    transforming into the frequency domain
// (examples: rolling mean/variance, rolling
    correlations or cross-correlations,
// rolling regression, windowed
    autocorrelation, rolling volatility).
// - "simple arithmetic", if the primary
    strategy consisted of direct,
    non-model-based
// summary calculations without statistical
    inference or signal analysis (examples:
// global mean/median/variance, counts, ratios,
     totals, expanding statistics, basic
// differencing).
```

```
// - "other", if none of the above fits.
"core_strategy":
  enum["statistical test", "spectral analysis",
     "curve fitting", "windowed/rolling
    statistics", "simple arithmetic", "other"
    ],
"core_strategy_expl": string,

"methodological_problems": {
  // "conceptual_misunderstanding" should be
    true if the agent demonstrates a
    fundamental
  // misunderstanding of key concepts relevant
     to the question.
  "conceptual_misunderstanding": bool,
  "conceptual_misunderstanding_expl": optional
    [string],

  // "wrong_core_problem_solving_strategy"
    should be true if the strategy chosen by
    the
  // agent ("core_problem_solving_strategy")
    was not suitable for answering the
    question.
  // For example, if the model decided to use
    curve fitting, when using a statistical
  // test would have been better.
  "wrong_core_problem_solving_strategy": bool,
  "wrong_core_problem_solving_strategy_expl":
  optional[string],

  // "wrong_method_within_strategy" should be
    true if the strategy chosen by the agent
  // ("core_problem_solving_strategy") was
    suitable for answering the question, but
  // failed due to the specific method chosen
    within the strategy. For example, the
    model
  // selected an incorrect statistical test,
    heuristic, threshold, or algorithmic
  // variant.
  "wrong_method_within_strategy": bool,
  "wrong_method_within_strategy_expl":
    optional[string],

  // "unsupported_assumption" should be true
    if for the method that the agent chose, it
  // made an unjustified assumption about the
    data or the options. For example, assuming
  // stationarity without evidence.
  "unsupported_assumption": bool,
  "unsupported_assumption_expl": optional[
    string],

  // "implementation_errors" should be true if
     there were coding or algorithmic mistakes
  // that affect the validity of the result.
    Code execution errors are that caused the
  // code to fail are not included in this
    category.
  "implementation_errors": bool,
  "implementation_errors_expl": optional[
    string],

  // "incorrect_result_interpretation" should
    be true if the agent incorrectly
    interprets
  // the result of any of his computations.
    For example, the model wrongly conclude
    that
  // the difference is statistically
    significant.
  "incorrect_result_interpretation": bool,
  "incorrect_result_interpretation_expl":
    optional[string],
```

877

```
  // "insufficient_evidence_guess" should be
    true if the agent answered whilst not
    having
  // gathered sufficiently convincing evidence.

  "insufficient_evidence_guess": bool,
  "insufficient_evidence_guess_expl": optional
    [string]
},

// This section is primarily meant for agents
    that had access to a code interpreter
// tool.
"code_problems": {
  // "code_result" should be one of the
    following options, indicating the overall
    success
  // of the code the model produced (if any):
  // - "success", if the agent managed to
    obtain the data it intended. If the code
  // initially failed due to errors, but the
    agent managed to fix the errors, it still
  // counts as a success.
  // - "partial failure", if the agent failed
    to obtain some of the desired data, but
  // managed to get at enough results to
    produce an answer. For example if the code
  // failed after printing sufficient evidence.

  // - "complete failure", if the agent failed
     to obtain the results and gave up or ran
  // out of turns.
  // - "no code", if the agent didn't produce
    any code.
  "code_result":
  enum["success", "partial failure", "complete
     failure", "no code"],
  "code_result_expl": optional[string],

  // "tool_usage_trouble" should be true if
    the agent had some trouble following
  // instructions to use the tool correctly.
    This mainly includes cases where the agent
  // generated an improper coding tool call or
     had trouble calling the tool properly.
  // Regular errors and exceptions that caused
     the code to fail are not included in this
  // category.
  "tool_usage_trouble": bool,
  "tool_usage_trouble_expl": optional[string],

  // "other" should be true if there was some
    other issue with the code that does not
  // fall into any of the other categories.
  "other": bool,
  "other_expl": optional[string]
},

"other_problems": {
  // "reasoning_answer_mismatch" should be
    true if the agent settled on one answer in
     the
  // thinking trace and then answered
    differently.
  "reasoning_answer_mismatch": bool,
  "reasoning_answer_mismatch_expl": optional[
    string],

  // "reasoning_tool_usage_mismatch" should be
     true if the agent claims it will use the
  // coding tool (such as saying "let's
    compute", or crafting a code solution in
    the
  // thinking trace) but then goes straight to
     answering.
  "reasoning_tool_usage_mismatch": bool,
```

878

```
  "reasoning_tool_usage_mismatch_expl":
    optional[string],

  // "hallucinated_values_in_reasoning" should
     be true if the agent introduces numerical
  // values that do not appear in the data or
    the code output. Reasonable rounding does
  // not count.
  "hallucinated_values_in_reasoning": bool,
  "hallucinated_values_in_reasoning_expl":
    optional[string]
}
},
```

Figure 8: Prompt for the LLM judge. The model receives the full conversation trace of the agent, including the question, the reasoning trace, the code (if applicable), and the final answer, along with the correct answer. The model is then asked to fill in a structured JSON questionnaire following the taxonomy described in Section 4.2. Note that we abbreviate `explanation` to `expl` in the field names for visualization purposes.

C Constructing the Error Taxonomy

We built an initial set of error categories by running two rounds of Gemini 3 Pro inference on a subset of incorrect `gpt-oss-120b` outputs. In the first round, Gemini 3 Pro received each incorrect trace paired with the correct answer and produced a short description of where the agent went wrong (Figure 9). In the second round, we fed all those descriptions into a single prompt asking Gemini to cluster them into fewer than ten categories and assign one to each entry (Figure 10).

Gemini returned seven preliminary categories:

1. **Skipping the Coding Phase:** The agent bypassed the coding tool or failed to execute planned code, relying on guesses instead.
2. **Statistical Misinterpretation & Methodological Errors:** Wrong statistical test, misread p-values or R^2, or failure to account for stationarity and distribution properties.
3. **Visual/Data Hallucination (Manual Inspection):** The agent described values and patterns from raw JSON without computing them.
4. **Conceptual & Semantic Misunderstandings:** Misunderstood core time series concepts (lag, amplitude, variance) or the specific definitions given in the problem.
5. **Coding Logic & Implementation Errors:** Buggy code, wrong library functions, or incorrectly implemented formulas.
6. **Flawed Heuristics & Rigid Thresholds:** Arbitrary numerical thresholds (e.g., 10^{-6} tolerance, 0.05 margin) producing incorrect binary decisions.
7. **Reasoning–Output Mismatch & Process Failures:** The agent ignored its own correct reasoning or code output when selecting the final answer.

These categories informed the failure mode section of the final taxonomy (Table 3). We expanded and restructured the section manually, adding the strategy and code problem categories.

```
You are given a conversation with a coding LLM
agent. In the conversation, the agent attempts
to answer a question by repeatedly producing
python code to programmatically query about
the relevant time series. The agent answered
wrong. The correct answer was "{data[correct]}".
Analyze where the error happened.
Answer in plain text. Do not produce more than
2 paragraphs.
=== Agent's Conversation ===
{data[conv]}
```

Figure 9: Prompt for per-trace error analysis. Each incorrect model output is analyzed individually.

```
{analysis 1}
-
(...)
-
{analysis n-1}
-
{analysis n}
======
Each chunk of text represents an analysis of
where an LLM coding agent made an error in
answering a question about time series. Cluster
all these errors into less than 10 error
categories. Then output a list assigning each
entry the error type. If an entry fits multiple
error types, select the most relevant one.
```

Figure 10: Prompt for clustering the per-trace descriptions into error categories.

D Selecting and Validating $\mathcal{L}_{\text{judge}}$

Development set. We build the development set from the configuration space $\mathcal{C} = \mathcal{D} \times \mathcal{S} \times \mathcal{M}$, where:

$$\mathcal{D} = \{\textsc{TimeSeriesExam}, \text{TSFU}\},$$
$$\mathcal{S} = \{\mathcal{M}_{\text{direct}}, \mathcal{M}_{\text{hybrid}}, \mathcal{M}_{\text{code}}\},$$
$$\mathcal{M} = \{\text{gpt-oss-120b}, \text{Qwen3-Next}\},$$

giving $|\mathcal{C}| = 12$. We sample 3 examples per configuration for 36 total. Each configuration's triple contains either 2 correct + 1 incorrect or 1 correct + 2 incorrect answers, chosen with equal probability, keeping the set roughly balanced. Since

Judge configuration	Accuracy
`qwen3-next-80b` (single-phase)	86.5%
`qwen3-next-80b` (two-phase)	88.1%
`gpt-5.4-mini-2026-03-17`	89.3%
`gpt-oss-120b` (no system msg.)	89.3%
`gpt-oss-120b` (system msg.)	**90.3%**

Table 6: Agreement of each candidate judge configuration with manual labels on the 36-example development set.

Qwen3-Next was not evaluated on the full TSFU dataset, the three TSFU × Qwen3-Next configurations were sampled from a separate pool of 60 randomly selected TSFU questions. No question appears more than once.

Judge configurations. We test five judge setups on the development set, all applying the taxonomy from Section 4.2:

- **Qwen3-Next (single-phase):** the model generates the required JSON output directly. Constrained decoding prevents it from emitting reasoning tokens, so judgments lack explanations.
- **Qwen3-Next (two-phase):** the model first answers in free-form text, then rewrites its response into the JSON schema in a second turn, recovering reasoning at the cost of an additional inference call.
- **gpt-oss-120b (no system message):** the annotation task is specified entirely in the user message.
- **gpt-oss-120b (system message):** the same model with a custom system message priming it as an expert annotator.
- **gpt-5.4-mini:** an alternative commercial model, helps to better ground the judge capabilities.

Results. Table 6 reports agreement with manual labels for each configuration. `gpt-oss-120b` with a system message scores highest at 90.3%, while Qwen3-Next single-phase scores lowest at 86.5%. The accuracy gap between the models is small overall. We select the best performing configuration `gpt-oss-120b` with a system message as $\mathcal{L}_{\text{judge}}$.

E Additional Results

Tables 7 and 8 break down accuracy by question category. Table 9 shows the distribution of strategy choices across domains. Table 10 shows the token usage statistics across all setups.

category	$\mathcal{M}_{\text{code}}$	$\mathcal{M}_{\text{direct}}$	$\mathcal{M}_{\text{hybrid}}$
`gpt-oss-120b`			
similarity analysis	70.0%	65.0%	**75.8%**
pattern recognition	75.7%	69.1%	**80.9%**
causality analysis	65.3%	45.8%	**75.0%**
noise understanding	71.4%	76.2%	**81.0%**
anolmaly detection	55.6%	57.4%	**70.4%**
`qwen3-next-80b`			
similarity analysis	60.8%	65.8%	65.8%
pattern recognition	59.9%	66.0%	70.7%
causality analysis	69.4%	61.1%	68.1%
noise understanding	65.5%	67.9%	75.0%
anolmaly detection	42.6%	50.9%	56.5%

Table 7: Per-category accuracies on TIMESERIESEXAM for both models and all three setups. The best score for each category across both models and all three setups is in bold.

	`gpt-oss-120b`		
category	$\mathcal{M}_{\text{code}}$	$\mathcal{M}_{\text{direct}}$	$\mathcal{M}_{\text{hybrid}}$
fixed correlation	**79.5%**	72.2%	78.8%
lagged correlation	**42.5%**	30.0%	37.5%
anomalies	65.0%	74.0%	**75.0%**
seasonality	63.0%	56.0%	**72.5%**
fat tail	**86.5%**	69.0%	85.0%
stationarity	46.0%	28.0%	**51.5%**
structural break	44.0%	**52.0%**	49.5%
trend	**96.5%**	88.0%	**96.5%**
volatility	27.5%	15.0%	**31.0%**

Table 8: Per-category accuracies on TSFU for all three setups. Only `gpt-oss-120b` results are shown; `qwen3-next-80b` was not evaluated on the full TSFU dataset (see Section 3.4). The best score in each row is in bold.

	simple arith.			stat. test			rolling stat.			curve fitting			spectral analysis			other			*n*
Domain	d	c	h	d	c	h	d	c	h	d	c	h	d	c	h	d	c	h	
TimeSeriesExam																			
Anomaly detection	67	40	63	0	12	7	0	14	6	0	9	13	0	12	8	33	13	4	637
Pattern recognition	80	37	42	0	19	17	0	4	4	0	24	24	0	10	10	20	7	3	2,171
Similarity	81	30	33	0	21	22	0	0	3	0	30	29	0	18	12	19	1	1	718
Noise understanding	69	26	32	0	38	40	0	11	10	1	10	10	0	9	8	30	7	1	504
Causality analysis	73	3	17	1	29	29	0	67	54	0	1	0	0	0	0	27	0	0	430
TSFU																			
Outliers	79	73	95	0	1	2	0	26	3	0	0	0	0	0	0	21	0	0	599
Trend	91	0	0	0	78	55	0	0	0	0	22	46	0	0	0	8	0	0	600
Statistical property	97	93	80	0	7	20	0	0	0	0	0	0	0	0	0	3	0	0	598
Correlation	96	25	15	0	75	85	0	0	0	0	0	0	0	0	0	4	0	0	1,198
Lagged correlation	86	3	20	0	83	64	0	14	15	0	0	0	0	0	0	14	0	0	597
Seasonality	72	16	18	0	0	0	0	0	2	0	0	0	0	84	78	28	0	1	599
Structural break	80	9	2	0	76	88	0	16	9	0	0	0	0	0	0	20	0	0	600
Stationarity	94	10	7	0	78	81	0	1	3	0	7	6	0	4	2	6	0	0	600
Volatility	82	0	0	0	0	0	2	100	100	0	0	0	0	0	0	16	0	0	600

Table 9: Distribution of core problem-solving strategies per problem domain, split by agent setup (dir. = $\mathcal{M}_{\text{direct}}$, code = $\mathcal{M}_{\text{code}}$, hyb. = $\mathcal{M}_{\text{hybrid}}$). Each cell shows the percentage of answers within the given domain and setup, rounded to the nearest integer; within each setup, the dominant strategy is shown in bold. n = total number of judge-annotated answers in that domain across all models and setups.

Model	**Setup**	**Data**	**Mean**	**Median**	**Max**	**Cut off (%)**
`gpt-oss-120b`	$\mathcal{M}_{\text{direct}}$	TSE	13,340	9,933	30,040	26.8
`gpt-oss-120b`	$\mathcal{M}_{\text{code}}$	TSE	4,036	2,668	30,997	4.2
`gpt-oss-120b`	$\mathcal{M}_{\text{hybrid}}$	TSE	4,693	2,459	46,622	8.2
`gpt-oss-120b`	$\mathcal{M}_{\text{direct}}$	TSFU	7,900	3,825	30,172	5.8
`gpt-oss-120b`	$\mathcal{M}_{\text{code}}$	TSFU	4,335	2,737	48,540	<0.1
`gpt-oss-120b`	$\mathcal{M}_{\text{hybrid}}$	TSFU	3,719	2,178	43,251	0.1
`qwen3-next`	$\mathcal{M}_{\text{direct}}$	TSE	15,983	10,068	275,925	3.4
`qwen3-next`	$\mathcal{M}_{\text{code}}$	TSE	14,115	7,820	123,674	1.2
`qwen3-next`	$\mathcal{M}_{\text{hybrid}}$	TSE	16,730	11,219	337,976	2.5

Table 10: Per-question output token statistics by model, setup, and dataset. Mean and median reflect the typical generation budget per question; the maximum captures the longest single conversation. "Cut off" is the percentage of questions whose answer was explicitly marked as truncated in the logged outputs. For $\mathcal{M}_{\text{code}}$ and $\mathcal{M}_{\text{hybrid}}$, token counts accumulate across all turns of the multi-turn conversation (code generation, execution feedback, and the final answer).