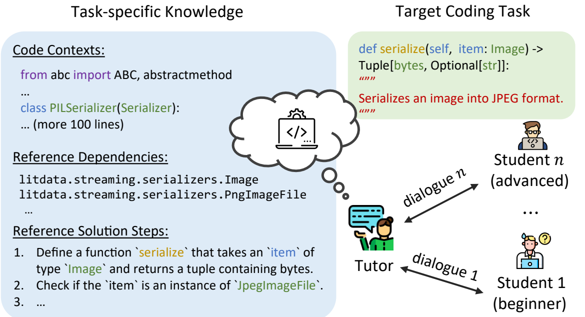
Training Turn-by-Turn Verifiers for Dialogue Tutoring Agents: The Curious Case of LLMs as Your Coding Tutors

Jian Wang[1,2*] **Yinpei Dai**[2] **Yichi Zhang**[2]
Ziqiao Ma[2] **Wenjie Li**[1] **Joyce Chai**[2]
[1]The Hong Kong Polytechnic University [2]University of Michigan
`jian-dylan.wang@connect.polyu.hk` `cswjli@comp.polyu.edu.hk`
`{daiyp,zhangyic,marstin,chaijy}@umich.edu`

Abstract

Intelligent tutoring agents powered by large language models (LLMs) have been increasingly explored to deliver personalized knowledge in areas such as language learning and science education. However, their capabilities in guiding users to solve complex real-world tasks remain underexplored. To address this limitation, in this work, we focus on *coding tutoring*, a challenging problem that requires tutors to proactively guide students towards completing predefined coding tasks. We propose a novel agent workflow, **Tra**ce-and-**Ver**ify (**TRAVER**), which combines knowledge tracing to estimate a student's knowledge state and turn-by-turn verification to ensure effective guidance toward task completion. We introduce **DICT**, an automatic evaluation protocol that assesses tutor agents using controlled student simulation and code generation tests. Extensive experiments reveal the challenges of coding tutoring and demonstrate that TRAVER achieves a significantly higher success rate. Although we use code tutoring as an example in this paper, our approach can be extended beyond coding, providing valuable insights into advancing tutoring agents for human task learning.[1]



```
from abc import ABC, abstractmethod
...
class PILSerializer(Serializer):
... (more 100 lines)
```

```
litdata.streaming.serializers.Image
litdata.streaming.serializers.PngImageFile
...
```

1. Define a function `serialize` that takes an `item` of type `Image` and returns a tuple containing bytes.
2. Check if the `item` is an instance of `JpegImageFile`.
3. ...

```
def serialize(self,  item: Image) ->
Tuple[bytes, Optional[str]]:
"""
Serializes an image into JPEG format.
"""
```

Figure 1: An illustration of coding tutoring, where a tutor aims to proactively guide students toward completing a target coding task while adapting to students' varying levels of background knowledge.

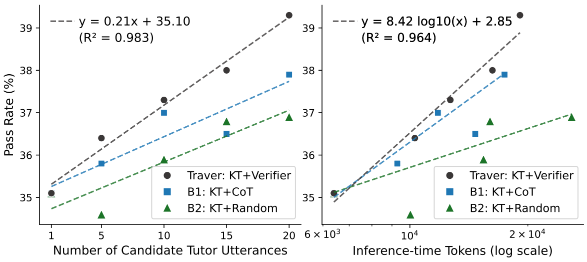


Figure 2: TRAVER with the trained verifier shows inference-time scaling for coding tutoring (detailed in §5.4). *Left*: Performance vs. sampled candidate utterances per turn. *Right*: Performance vs. total tokens consumed per tutoring session.

1 Introduction

Tutoring has long been recognized as one of the most effective methods for enhancing human learning outcomes and addressing educational disparities (Hill et al., 2005). By providing personalized guidance to students, intelligent tutoring systems (ITS) have proven to be nearly as effective as human tutors in fostering deep understanding and skill acquisition, with research showing comparable learning gains (VanLehn, 2011; Rus et al., 2013). More recently, the advancement of large language models (LLMs) has offered unprecedented opportunities to replicate these benefits in tutoring agents (Dan et al., 2023; Jin et al., 2024; Chen et al., 2024), unlocking the enormous potential to solve knowledge-intensive tasks such as answering complex questions or clarifying concepts.

Previous research has extensively explored tutoring in *knowledge delivery*, including language learning (Swartz and Yazdani, 2012; Stasaski et al., 2020), mathematical reasoning (Demszky and Hill, 2023; Macina et al., 2023), and scientific concept education (Yuan et al., 2024; Yang et al., 2024b). Most aim to enhance students' understanding of target knowledge by employing pedagogical strategies such as recommending exercises (Deng et al.,

---

[*] This work was conducted while Jian Wang was a visiting PhD student at the University of Michigan.

[1] Code and data are available at `https://github.com/iwangjian/Coding-Tutor`.

Findings of the Association for Computational Linguistics: ACL 2025, pages 12416–12436
July 27 - August 1, 2025 ©2025 Association for Computational Linguistics

2023) or selecting teaching examples (Ross and Andreas, 2024). However, these approaches fall short in broader *task tutoring* situations demanding both understanding and practical application of specific pieces of knowledge to solve real-world, goal-driven problems. Such scenarios require tutors to proactively guide people toward completing targeted tasks (e.g., coding). Furthermore, the tutoring outcomes are challenging to assess since targeted tasks can often be completed with open-ended solutions.

To bridge this gap, we introduce **coding tutoring**, a promising yet underexplored task for LLM agents. As illustrated in Figure 1, the tutor is provided with a target coding task and task-specific knowledge (e.g., cross-file dependencies and reference solutions), while the student is given only the coding task. The tutor does not know the student's prior knowledge about the task. Coding tutoring requires the tutor to proactively guide the student toward completing the target task through dialogue. This is inherently a goal-oriented process where tutors guide students using task-specific knowledge to achieve predefined objectives. Effective tutoring requires personalization, as tutors must adapt their guidance and communication style to students with varying levels of prior knowledge.

Developing effective tutoring agents is challenging because off-the-shelf LLMs lack grounding to task-specific knowledge and interaction context. Specifically, tutoring requires *epistemic grounding* (Tsai and Roth, 2016), where domain expertise and assessment can vary significantly, and *communicative grounding* (Chai et al., 2018), necessary for proactively adapting communications to students' current knowledge. To address these challenges, we propose the **Tra**ce-and-**Ver**ify (**TRAVER**) agent workflow for building effective LLM-powered coding tutors. Leveraging knowledge tracing (KT) (Corbett and Anderson, 1994; Scarlatos and Lan, 2024), TRAVER explicitly estimates a student's knowledge state at each turn, which drives the tutor agents to adapt their language to fill the gaps in task-specific knowledge during utterance generation. Drawing inspiration from value-guided search mechanisms (Lightman et al., 2024; Wang et al., 2024a; Zhang et al., 2024), TRAVER incorporates a turn-by-turn reward model as a verifier to rank candidate utterances. By sampling more candidate tutor utterances during inference (see Figure 2), TRAVER ensures the selection of optimal utterances that prioritize goal-driven guidance and advance the tutoring progression effectively. Furthermore, we present **Di**alogue for **C**oding **T**utoring (**DICT**), an automatic protocol designed to assess the performance of tutoring agents. DICT employs code generation tests and simulated students with varying levels of programming expertise for evaluation. While human evaluation remains the gold standard for assessing tutoring agents, its reliance on time-intensive and costly processes often hinders rapid iteration during development. By leveraging simulated students, DICT serves as an efficient and scalable proxy, enabling reproducible assessments and accelerated agent improvement prior to final human validation.

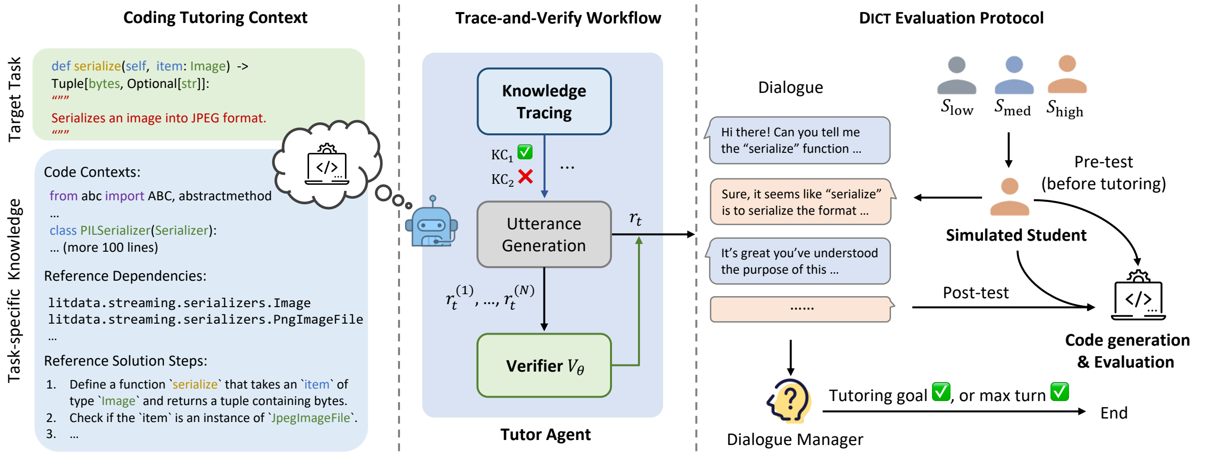
Through extensive experiments, we show that agents developed by TRAVER consistently demonstrate higher success rates in guiding students to complete target coding tasks compared to baseline methods. We present detailed ablation studies, human evaluations, and an inference time scaling analysis, highlighting the transferability and scalability of our tutoring agent workflow.

2 Problem Definition

We formulate **coding tutoring** as an interactive dialogue process between a **tutor** and a **student**, where the goal is to help the student implement a working solution that passes predefined unit tests for a target coding task.

Formally, the tutor is assigned a coding task $\mathcal{T}$ that consists of a function signature and a requirement description outlining the desired functionality. The tasks are repository-level, which require an understanding of multiple interdependent files within the codebase to implement a correct solution. The tutor has access to task-specific knowledge $\mathcal{K}$, which includes (i) *Code Contexts*: Contextual code snippets surrounding the desired code, which help the tutor show examples when necessary; (ii) *Reference Dependencies*: Cross-referenced elements such as intra-class, intra-file, and cross-file dependencies, along with their corresponding descriptions (e.g., docstrings), which involve key knowledge for completing the desired code; and (iii) *Reference Solution Steps*: Key steps required to complete the target task, describing using natural languages.

The student is given the task $\mathcal{T}$ and possesses some subset of $\mathcal{K}$ as their prior knowledge, but the tutor remains unaware of which specific concepts



```
def serialize(self,  item: Image)  ->
Tuple[bytes, Optional[str]]:
"""
Serializes an image into JPEG format.
"""
```

Code Contexts:

```
from abc import ABC, abstractmethod
…
class PILSerializer(Serializer):
… (more 100 lines)
```

Reference Dependencies:

```
litdata.streaming.serializers.Image
litdata.streaming.serializers.PngImageFile
…
```

Reference Solution Steps:

1. Define a function \`serialize\` that takes an \`item\` of type \`Image\` and returns a tuple containing bytes.
2. Check if the \`item\` is an instance of \`JpegImageFile\`.
3. …

Hi there! Can you tell me the "serialize" function …

Sure, it seems like "serialize" is to serialize the format …

It's great you've understood the purpose of this …

……

Figure 3: Overview of our work for developing coding tutoring agents. *Left*: The context of the coding tutoring problem. *Middle*: Trace-and-Verify (TRAVER) workflow. *Right*: DICT evaluation protocol.

or dependencies the student has already mastered. The goal of the tutor is to guide the student, regardless of his or her background, toward solving the task $\mathcal{T}$ through multi-turn interactions.

3 Trace-and-Verify Agent Workflow

We propose **Tra**ce-and-**Ver**ify (**TRAVER**), an effective workflow for developing tutor agents (see the middle part in Figure 3). TRAVER integrates two key components: (i) explicit tracing of a student's knowledge state and (ii) utterance decoding guided by a verifier model for turn-by-turn verification.

3.1 Adapting to Student's Knowledge via Knowledge Tracing

Effective tutoring requires bridging the gap between a student's prior knowledge and the skills needed to solve the target coding task. To address this, we employ knowledge tracing (KT) (Corbett and Anderson, 1994; Abdelrahman et al., 2023; Scarlatos and Lan, 2024) to estimate the student's knowledge state at each dialogue turn. Specifically, we represent task-specific knowledge $\mathcal{K}$ as a set of knowledge components (KCs) $\{\text{KC}_1, \text{KC}_2, \ldots, \text{KC}_K\}$, where each KC is either a reference dependency or a solution step. At the t-th turn, the tutor agent explicitly assesses the student's belief of each KC using texts, based on the dialogue context $\mathcal{C}_t$ and the estimated belief B_{t-1} at the previous turn. The current belief B_t indicates how many KCs the student has understood. With this estimation, the tutor is further prompted to focus more on missing KCs and generate utterances that address the student's knowledge gaps. The detailed prompt template is provided in Appendix E.

3.2 Utterance Decoding via Turn-by-Turn Verification

Based on the KT outcomes, the tutor agent aims to generate high-quality utterances that advance the tutoring process. However, LLMs often struggle to determine which utterances effectively guide students toward task completion. Drawing inspiration from value-guided search approaches (Lightman et al., 2024; Wang et al., 2024a; Zhang et al., 2024), we address this with a turn-by-turn verifier. The verifier V_θ evaluates the quality of potential tutor responses by producing a reward score $v_t \in [0, 1]$ based on three inputs: the target task $\mathcal{T}$, current dialogue context $\mathcal{C}_t$, and a candidate tutor utterance r_t at turn t. To select the optimal response, we generate N candidate utterances through parallel sampling and choose the one that receives the highest reward score from the verifier.

The core of the verifier V_θ is the turn-based reward v_t at t-th turn, which should reflect (i) the cumulative progress made in the previous turns and (ii) the current turn's contribution to achieving the overall tutoring goal. Hence, v_t can be iteratively defined as:

$$v_t = \max(v_{t-1} + w_{r_t}, 0) \; (v_0 = 0, t \in [1, T]) \quad (1)$$

where w_{r_t} is the weighted reward quantifying the contribution of the current turn to the overall goal, T denotes the total number of turns. To compute w_{r_t}, we introduce the concept of guiding distance $d_t = T - t$, which measures the remaining turns until the goal or end of the interaction. The weighted reward is then calculated as:

$$w_{r_t} = \frac{1 - v_{t-1}}{d_t + 1}(2o_{s_t} - 1) \quad (2)$$

where $o_{s_t} \in \{0, 1\}$ is a binary outcome indicating whether the tutor's t-th utterance contributes to the student's eventual successful completion of the target task. As the dialogue progresses, the guiding distance d_t decreases, leading to larger weighted rewards for later turns. This design ensures the turn-based reward v_t remains bounded while appropriately weighting the importance of each turn based on its proximity to the goal.

We train the verifier V_θ using mean squared error (MSE) loss over n samples:

$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^{n} \sum_{t=1}^{T_i} \left(V_\theta([\mathcal{T}^{(i)}; \mathcal{C}_t^{(i)}; r_t^{(i)}]) - v_t^{(i)} \right)^2 \tag{3}$$

where T_i denotes the number of turns for the i-th sample, θ denotes the trainable parameters.

During inference, the verifier serves as a plug-and-play module, which chooses the utterance with the highest reward from candidate utterances generated by parallel sampling at each turn, promoting progression toward tutoring task completion.

4 The DICT Evaluation Protocol

One key challenge in developing tutoring agents is the lack of robust evaluation methods. While human evaluation is essential, its high cost, time requirements, and complexity make it impractical for scalable benchmarking. To address this limitation, we present **Di**alogue for **C**oding **T**utoring (**DICT**), an automatic protocol for evaluating tutor agents. The overview of DICT is shown in Figure 3. Our protocol employs LLMs to simulate students with varying levels of programming knowledge. First, tutors engage in multi-turn dialogues with students to tutor the task. Then, students demonstrate their learning outcome by implementing the solution code. We evaluate tutoring effectiveness through automated unit tests of the student-generated code. This automated approach enables controlled, reproducible, and scalable evaluations of tutoring agents.

4.1 Controlled Student Simulation

To evaluate how well tutors can adapt their strategies to students with varied prior knowledge, we simulate students of three knowledge levels: (1) **Low-level**: Students access no prior knowledge from $\mathcal{K}$. They represent beginners with no familiarity with the target task. (2) **Medium-level**: These students are assigned a proportion (e.g., 50%) of the *reference dependencies* by random sampling, denoting that they have partial knowledge required for completing the task. (3) **High-level**: In addition to partial *reference dependencies*, these students are also provided with the *code contexts*, indicating that they have more comprehensive knowledge with contextual guidance.

Pre-Test. We create students of all three levels using the same LLM simulator, varying only in their knowledge level. However, this raises a critical question: *do students with different knowledge levels actually demonstrate distinct performance in completing the coding tasks?* To validate this, we conduct a preliminary coding test where each simulated student attempts to generate code for the target task $\mathcal{T}$ before any tutoring intervention (see Appendix E for prompting details).

Metrics for Coding Test. Following previous studies (Austin et al., 2021; Li et al., 2024), we employ Recall@k and Pass@k as evaluation metrics to assess coding test performance. Recall@k measures the recall of reference dependencies in the generated programs. Specifically, students are asked to generate k programs per target task. For the i-th program, we extract its dependencies $\mathbb{P}_i$ using the Pyan parser (Pyan, 2023). We compare them with the reference dependencies $\mathbb{R}$ and compute the Recall@k as:

$$\text{Recall@}k = \mathop{\mathbb{E}}_{\text{Target Tasks}} \left[\max_{i \in [1,k]} \frac{|\mathbb{R} \cap \mathbb{P}_i|}{|\mathbb{R}|} \right] \tag{4}$$

where $|\cdot|$ denotes the number of elements of a set. Pass@k evaluates the functional correctness of the generated programs. After generating $n \geq k$ programs per task, we execute them in Python interpreters to count the number of correct programs c that pass all test cases. Pass@k is computed as:

$$\text{Pass@}k = \mathop{\mathbb{E}}_{\text{Target Tasks}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \tag{5}$$

where $\binom{\cdot}{\cdot}$ denotes the number of ways to choose a subset of elements (also known as the binomial coefficient). Our pre-test results reported in §5.3 show that the simulated students are effective.

4.2 Tutor-Student Interaction

As shown in Figure 3, we let an LLM-based tutor agent engage in a multi-turn dialogue with a chosen student, simulating a tutoring session. The tutor is initialized with the target coding task $\mathcal{T}$ and task-specific knowledge $\mathcal{K}$. We ask the tutor to initiate the tutoring and communicate with the student turn by turn. A key challenge is determining when to terminate the tutoring. While the tutor could self-determine, our preliminary experiments revealed a tendency for overconfidence, leading to premature termination. This issue arises because most tutors overlook gaps in the student's prior knowledge. For a robust comparison, we follow Wang et al. (2023) and introduce an LLM-powered dialogue manager (see Figure 3). Operating from a "God's perspective," the manager considers the dialogue context and all information from both the tutor and student, to decide whether the tutoring goal has been met. The tutoring terminates under one of two conditions: (i) The manager confirms that the tutoring goal is achieved; (ii) The dialogue reaches a predefined maximum of T turns.

4.3 Automatic Evaluation

Post-Test. To evaluate the effectiveness of tutor agents, we conducted a coding test after tutoring (referred to as the post-test). Given a target task $\mathcal{T}$ and a dialogue session $\mathcal{C} = \{s_t, r_t\}_{t=1}^{T}$ with a simulated student, we ask the student to generate code for fulfilling the task. However, assuming that all dialogue content is retained during the coding test may be unrealistic. According to cognitive load theory (CLT) (Miller, 1956; Sweller, 2011), human working memory has a limited learning capacity at one time, and exceeding this capacity can hinder learning. This cognitive load can be affected by the complexity of the target task or student engagement during the tutoring dialogue.

As a simple, practical alternative, we consider the student's cognitive load f_{CL} at each turn by assuming the information retained from the tutor's utterance to a maximum threshold. Specifically, if the tutor's utterance r_t exceeds M words, only the latest M words are retained; otherwise, the full utterance is kept. Our post-test is formulated as:

$$\mathcal{Y}_{\text{code}} = \text{LM}_{\text{Student}}([\mathcal{I}; \mathcal{T}; \{s_t, f_{\text{CL}}(r_t)\}_{t=1}^{T}]) \quad (6)$$

where $\mathcal{I}$ represents the instruction for code generation. s_t and r_t denote student and tutor utterances at t-th turn, respectively. A detailed template for the instruction can be found in Appendix E.

Evaluation Metrics. Based on the coding test, students' post-test performance after tutoring is defined as the **tutoring outcome** (**TO**), measured by **Recall** and **Pass**. They represent the averages of Recall@k and Pass@k for $k \in \{1, 3, 5, 10\}$. A higher Recall score indicates the tutor is more capable of leading the student to acquire the dependency knowledge for coding; a higher Pass score denotes a higher success rate of guiding the student in completing the target coding task.

Due to the difference in prior knowledge levels, we use the **tutoring outcome rate** ($\Delta\%$) to normalize and fairly evaluate the tutor's performance. This is calculated by relative improvement before and after tutoring:

$$\Delta\%\mathcal{M} = \left(\frac{\mathcal{M}_{\text{post-test}} - \mathcal{M}_{\text{pre-test}}}{\mathcal{M}_{\text{pre-test}}} \right) \times 100\% \quad (7)$$

where $\mathcal{M}$ denotes the metric for the coding test, which can be either Recall or Pass.

To further analyze the tutoring process, we propose the **tutoring outcome curve** (**TOC**). At each t-th turn, we ask simulated students to perform a post-test using the dialogue context up to that turn, i.e., $\{s_{\leq t}, r_{\leq t}\}$. The TOC is then plotted by tracking the Recall and Pass scores varying by turns. These curves exhibit how tutor agents guide students throughout the tutoring session.

5 Experiments

5.1 Experimental Setup

Benchmark. We adopt **EvoCodeBench** (Li et al., 2024) as the testbed due to its realistic repository-level Python coding tasks along with dependency annotations and repository contexts, providing a rich knowledge foundation for coding tutoring. We have 100 target coding tasks and split them equally into 5 folds, using 5-fold cross-validation for experiments. The detailed examples, statistics, and preprocessing are provided in Appendix A.1.

Student Simulator. Prior to tutoring, it is essential to ensure that simulated students have not been exposed to any target coding task. To avoid data contamination, we use Mixtral-8x7B-Instruct (Jiang et al., 2024) as the student simulator. This model's training data only includes content up to `2023-9`, whereas all coding tasks in EvoCodeBench are collected from the repositories created between `2023-10` and `2024-2`. Furthermore, this model has strong conversational and coding abilities, making it well-suited for student simulation. Since different simulators can influence tutoring outcomes, we also use GPT-4o (OpenAI,

Method	Backbone Model	Overall		Low-level		Med.-level		High-level	
		Recall / Δ%R	Pass / Δ%P	Δ%R	Δ%P	Δ%R	Δ%P	Δ%R	Δ%P
Pre-Test	–	$45.9_{\pm 2.8}$ / –	$21.2_{\pm 2.0}$ / –	–	–	–	–	–	–
Vanilla Instruct	Qwen2-7B-Instruct	$56.4_{\pm 1.6}$ / 22.8	$26.4_{\pm 3.4}$ / 24.5	99.4	69.8	10.1	37.5	3.0	-0.3
	Qwen2-72B-Instruct	$61.4_{\pm 1.9}$ / 33.8	$32.1_{\pm 7.0}$ / 51.4	131.8	128.2	13.7	53.7	11.5	21.4
	Llama-3.1-8B-Instruct	$63.6_{\pm 4.4}$ / 38.5	$31.1_{\pm 3.3}$ / 46.7	138.6	145.1	23.8	48.3	10.9	8.0
	Llama-3.1-70B-Instruct	$62.5_{\pm 4.0}$ / 36.0	$34.9_{\pm 5.4}$ / 65.0	127.4	160.6	21.5	71.0	11.7	24.9
	GPT-3.5-Turbo	$60.1_{\pm 4.0}$ / 31.0	$28.8_{\pm 3.6}$ / 35.9	110.9	130.1	21.6	37.7	6.9	-1.3
	GPT-4o	$64.2_{\pm 4.3}$ / 39.9	$38.7_{\pm 5.6}$ / 82.8	141.3	207.8	28.4	102.1	8.9	23.5
	o1-mini	$61.3_{\pm 2.1}$ / 33.4	$35.9_{\pm 1.4}$ / 69.4	129.5	159.1	19.3	101.2	6.9	16.6
Self-Refine	GPT-4o	$64.0_{\pm 4.9}$ / 39.5	$40.6_{\pm 3.7}$ / 91.7	143.0	221.4	23.1	118.6	11.8	26.2
TreeInstruct	GPT-4o	$64.3_{\pm 2.8}$ / 40.1	$39.8_{\pm 1.5}$ / 88.1	154.8	211.5	21.8	100.3	9.7	33.6
TRAVER (Ours)	Llama-3.1-70B-Instruct	$66.8_{\pm 1.3}$ / 45.5	$39.3_{\pm 6.9}$ / 85.7	164.5	206.9	23.4	104.1	**16.4**	28.4
	GPT-4o	**68.8**$_{\pm 3.7}$ / **49.8**	**43.7**$_{\pm 1.3}$ / **106.5**	**166.3**	**242.5**	**34.8**	**122.9**	15.8	**44.8**
Oracle	–	$74.0_{\pm 5.7}$ / 61.2	$51.9_{\pm 3.7}$ / 144.8	200.8	318.5	42.3	176.1	21.1	60.7

Table 1: Automatic evaluation results of various LLM-based tutor agents. "Δ%R" and "Δ%P" represent the tutoring outcome rates (TOR) in Recall and Pass, respectively.

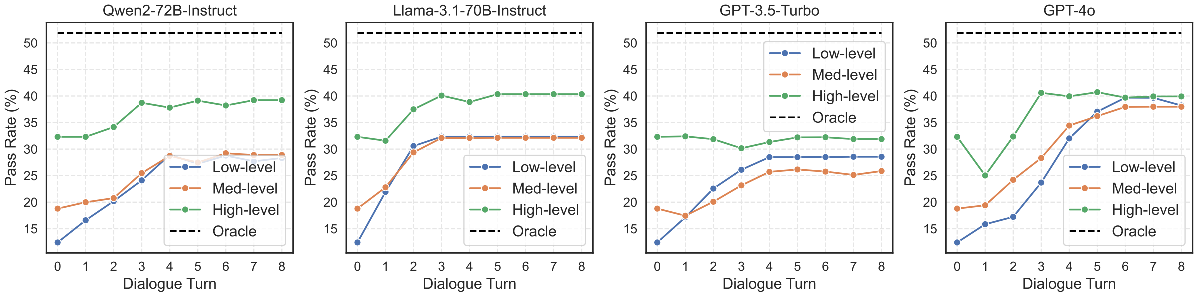


Figure 4: Tutoring outcome curves in Pass rate across various LLM-based tutors with Vanilla Instruct.

2024a) as the student simulator and report experimental results in Appendix B.

Backbone Models. We employ various backbone models to develop tutor agents, including: **Qwen2-Instruct** (Yang et al., 2024a) with 7B and 72B variants, **Llama-3.1-Instruct** (Dubey et al., 2024) with 8B and 70B variants, **GPT-3.5-Turbo** (OpenAI, 2022), **GPT-4o** (OpenAI, 2024a), and **o1-mini** (OpenAI, 2024b).

Baseline Methods We evaluate our TRAVER, against the following relevant baseline methods: (1) **Vanilla Instruct**: It directly instructs LLMs as tutors, as detailed in §4.2. (2) **Self-Refine** (Madaan et al., 2023): LLMs generate initial responses and iteratively refine these responses by providing natural language feedback to themselves. (3) **TreeInstruct** (Kargupta et al., 2024): It is a Socratic teaching method that estimates a student's knowledge and employs tree-based questioning to guide the student. (4) **Oracle**: It provides the full task-specific knowledge directly to each student during the post-test (see Eq. (6)).

Implementation Details. We implement the verifier in TRAVER using Mistral-7B (Jiang et al., 2023) with an additional linear layer. We utilize the synthesized dialogues from various backbone LLMs with vanilla instructions for the verifier, where the post-test results on Pass scores provide outcome reward labels. We adopt 5-fold cross-validation for training and evaluation. In the DICT evaluation protocol, the maximum number of turns T is set to 8. The cognitive load parameter M is set to 60 during the post-test. More details about training and inference are provided in Appendix A.2.

5.2 Experimental Results

How do various LLMs perform as tutor agents when provided with vanilla instructions? As shown in Table 1, various LLM-based tutors with vanilla instructions perform significantly inferior to the Oracle tutor, indicating clear limitations. Scaling the parameter size of open-source models like Qwen2 and Llama-3.1 generally improves performance. However, the large gaps in Pass and TOR-Pass scores suggest that simply using larger models

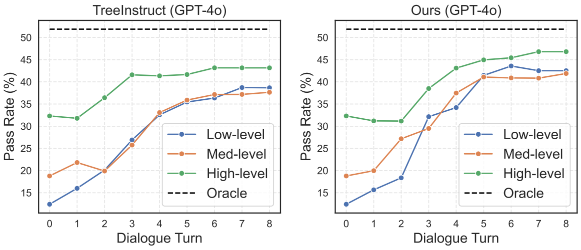


Figure 5: Comparison of tutoring outcome curves between the TreeInstruct and TRAVER (Ours).

Backbone Model	Recall / Δ%R	Pass / Δ%P
Llama-3.1-70B-Instruct	**66.8**$_{\pm 1.3}$ / **45.5**	**39.3**$_{\pm 6.9}$ / **85.7**
w/o KT	66.1$_{\pm 3.4}$ / 43.9	35.8$_{\pm 3.8}$ / 69.2
w/o Verifier	66.7$_{\pm 3.2}$ / 45.3	35.1$_{\pm 5.1}$ / 65.8
GPT-4o	**68.8**$_{\pm 3.7}$ / **49.8**	**43.7**$_{\pm 1.3}$ / **106.5**
w/o KT	65.9$_{\pm 2.2}$ / 43.5	41.7$_{\pm 2.7}$ / 97.1
w/o Verifier	67.8$_{\pm 0.8}$ / 47.7	39.8$_{\pm 3.2}$ / 88.2

Table 2: Ablation study results of our TRAVER.

is inadequate to guide students in successfully completing target coding tasks. These findings indicate that developing effective tutor agents requires not only detailed instructions but also mechanisms to facilitate tutoring outcomes in a structured way.

Another limitation that emerges from these results is the adaptability. As shown in Table 1, LLMs generally perform better with low-level than with high-level students. This discrepancy arises from the greater difficulty in adapting to higher-level students who require nuanced and targeted guidance. For example, Qwen2-7B-Instruct and GPT-3.5-Turbo show a decline in Pass rates for high-level students after tutoring (i.e., $\Delta\%\text{P} < 0$). Figure 4 illustrates this trend by tutoring outcome curves. Larger models like Qwen2-72B-Instruct and Llama3-3.1-70B-Instruct exhibit significant performance gaps between high-level students and others, even as tutoring dialogues progress. While GPT-4o demonstrates better adaptability, its tutoring outcomes early plateau from the third turn for high-level students, indicating diminishing returns. These results highlight the importance of enhancing tutor agents to adaptively guide different students.

Can TRAVER improve tutoring outcomes and better adapt to different students? As shown in Table 1, our TRAVER, built upon Llama-3.1-70B-Instruct, achieves notable improvements over Vanilla Instruct (e.g., from 34.9% to 39.3% in Pass rate). When compared to tutor agents built upon GPT-4o, TRAVER achieves the highest overall Recall and Pass rates. These results highlight that our approach is more effective in guiding students to successfully complete target coding tasks. More importantly, TRAVER exhibits substantial improvements across students with different levels, narrowing the gap with the Oracle tutor. The tutoring outcome curves in Figure 5 further illustrate that, regardless of students' prior knowledge levels, our method consistently improves the success rate of guiding students to achieve task completion.

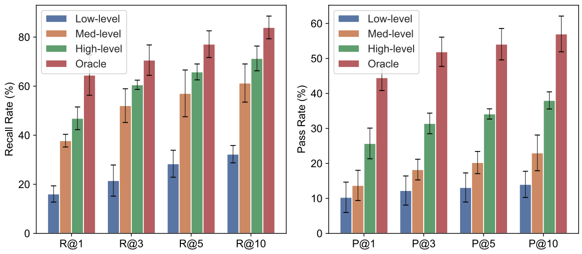


Figure 6: Pre-test performance of simulated students at different levels before tutoring.

Ablation study. We conduct an ablation study for TRAVER: (1) ***w/o knowledge tracing (KT)***, which removes the KT operation prior to tutor utterance generation; and (2) ***w/o verifier***, which omits the verifier V_θ used for verification. The results in Table 2 show that both the KT and verifier contribute to the overall performance. Notably, the absence of the verifier leads to a sharp decline in Pass and TOR-Pass, underscoring its critical role. The verifier improves the possibility of generated utterances that effectively advance the tutoring progress at each turn, thereby increasing the success rate in guiding students to complete the coding task.

5.3 Analysis of Simulated Students

Since the DICT evaluation protocol relies on the LLM-simulated students, it is crucial to ensure that these students, with varying levels of prior knowledge, align with discrepancies in completing coding tasks. To validate this, we conducted a pre-test for the simulated students before tutoring, as described in §4.1. Figure 6 shows that simulated low-, medium-, and high-level students exhibit significant performance differences in terms of Recall@k and Pass@k. Under a controlled setup, students at different levels demonstrate distinct abilities in completing target coding tasks. Therefore, our student simulation serves as a feasible proxy for real students, offering its advantages of scalability and cost-effectiveness for evaluating tutor agents.

Compared Methods	Proactivity			Adaptability			Coherence		
	Win (%)	Lose (%)	Tie (%)	Win (%)	Lose (%)	Tie (%)	Win (%)	Lose (%)	Tie (%)
TRAVER vs. Vanilla Instruct	**42.2**	26.7	31.1	**40.0**	33.3	26.7	24.4	**25.6**	50.0
TRAVER vs. Self-Refine	**38.9**	26.7	34.4	**41.1**	30.0	28.9	**27.8**	18.9	53.3
TRAVER vs. TreeInstruct	**34.4**	22.2	43.3	**37.8**	25.6	36.7	**28.9**	20.0	51.1

Table 3: Human evaluation results. For win and lose percentages, the higher value is bolded.

5.4 Inference-Time Scaling with Verifier

Using Llama-3.1-70B-Instruct as the backbone model, we vary the number of candidate tutor utterances per turn, i.e., N, within $\{1, 5, 10, 15, 20\}$ and ask the verifier to select the best one based on predicted rewards. We also evaluate two baselines: (i) Chain-of-Thought (CoT) (Wei et al., 2022) prompting, which performs zero-shot CoT reasoning (i.e., “Let’s think step by step and select the best response.”) to choose the most appropriate tutor utterance from a set of candidates at each turn (denoted as B1), and (ii) random selection (denoted as B2). As shown in Figure 2, TRAVER with the trained verifier improves the student’s Pass rate from 35.1% to 39.3% as N increases, outperforming both the random and CoT baselines while exhibiting stronger linear scaling. Additionally, in terms of total tokens (including input prompts and output responses) consumed per tutoring session, TRAVER achieves a better balance between tutoring performance and efficiency. These findings demonstrate that our TRAVER effectively enables inference-time scaling for tutoring agents.

5.5 Human Evaluation and Case Study

To evaluate the quality of tutor agents developed by different methods, we conducted a human evaluation for Vanilla Instruct, Self-Refine, TreeInstruct, and our TRAVER. We presented human evaluators with a pair of tutoring dialogues produced by two agents interacting with the same student. Evaluators were asked to determine which one is better from *proactivity*, *adaptability*, and *coherence*. Further details are provided in Appendix C.

Table 3 presents the evaluation results, with an average Fleiss’s kappa (κ) of 0.45, indicating moderate agreement among evaluators ($0.41 < \kappa < 0.60$). The results demonstrate that our TRAVER significantly outperforms the compared methods in proactivity and adaptability, while also matching or surpassing Vanilla Instruct in coherence. We provide several examples in Appendix D. To further illustrate the performance of our tutor agents, we provide several examples in Appendix D.

6 Related Work

Interactive Tutoring. As an advanced form of tutoring systems, interactive intelligent tutoring systems (ITSs) (Graesser et al., 2001; Rus et al., 2013; Liu et al., 2024c) can provide personalized feedback and adaptive learning experiences. They have been extensively explored across various educational domains for knowledge delivery, such as language learning (Swartz and Yazdani, 2012; Stasaski et al., 2020; Caines et al., 2020; Kwon et al., 2024), mathematical reasoning (Demszky and Hill, 2023; Macina et al., 2023; Wang et al., 2024b; Liu et al., 2024a), and scientific concept education (Yuan et al., 2024; Yang et al., 2024b). These studies mainly focus on enhancing students’ understanding of specific pieces of knowledge, using pedagogical strategies such as designing exercises (Deng et al., 2023; Wang et al., 2022; Lu et al., 2023), selecting teaching examples (Ross and Andreas, 2024), and remediating student reasoning errors (Daheim et al., 2024). Furthermore, the effectiveness of these approaches is often measured using closed-form assessments, such as question-answering (Yuan et al., 2024) or multiple-choice tests (Macina et al., 2023). Instead of focusing on specific knowledge delivery, we explore task-level tutoring, using coding tutoring as a representative example. This domain requires students to engage in open-ended code generation to evaluate tutoring effectiveness.

LLM-based Tutoring Agents. The rapid growth of large language models (LLMs) has expanded ITSs into tutoring agents (Yu et al., 2024). For instance, early efforts such as EduChat (Dan et al., 2023) introduced an educational chatbot for online tutoring, while CHATTUTOR (Chen et al., 2024) equipped tutor agents with course planning and adaptive quizzes to facilitate long-term interactions. As coding has emerged as a crucial domain for validating complex reasoning ability (Jimenez et al., 2024), AlgoBo (Jin et al., 2024), a recent LLM-based teachable agent, was developed to enhance students’ coding skills. We observe that existing

LLM agents primarily play a *reactive* role, focusing on answering questions or clarifying concepts. In comparison, our coding tutoring is both goal-driven and personalized, requiring agents to *proactively* guide students toward completing targeted coding tasks while adapting to diverse levels of knowledge priors. Our work presents a novel method that empowers tutor agents to address these challenges.

Inference-Time Adaptation of LLMs To enhance the controllability of language generation in complex tasks, prior work has investigated guided decoding (Dathathri et al., 2020; Chaffin et al., 2022) during inference. More recently, a notable line of research (Lightman et al., 2024; Li et al., 2023; Wang et al., 2024a; Pan et al., 2024) has employed verifier models complemented with search algorithms to guide LLMs for agentic reasoning. These methods typically focus on static tasks, often overlooking interactive scenarios. To address multi-turn interactions (Wang et al., 2024c), we introduce a turn-by-turn verifier that dynamically evaluates tutoring progress over time.

7 Conclusion and Future Work

This work explores the potential of LLMs as task-tutoring agents, using coding tutoring as a representative example. We propose TRAVER, an effective workflow that incorporates knowledge tracing and turn-by-turn verification, to tackle key challenges in coding tutoring. While this work focuses on coding tutoring as an example, the proposed method extends beyond coding to other task-tutoring scenarios. We further introduce DICT, a novel evaluation protocol combining student simulation and coding tests to assess tutor performance. Such automated evaluation is critical for developing task-tutoring agents as it supports a systematic development and evaluation cycle. Although it's outside the scope of this paper, the best-performing agent from the automated evaluation can be further assessed through studies with real human students in the future.

Limitations

In this work, we employed LLMs to simulate students at different knowledge levels, serving as a proxy for real-world learners. While these simulated students offer convenience and scalability, their representation of the human learning process is inherently limited. The role-playing behavior may differ from that of actual students in tutoring scenarios. Future research should focus on improving the reliability of student simulation to better align with real-world human learning.

In addition, the tutor agents were primarily evaluated by interacting with simulated students in our experimental setup. It remains unclear how these agents would perform when guiding humans toward completing target coding tasks. An important direction for future work is to extend our evaluation protocol by incorporating human-in-the-loop assessments, where tutor agents interact directly with actual students with necessary programming backgrounds. This would offer deeper insights into the practical effectiveness of the developed agents in real-world settings.

Ethics Statement

We strictly follow the protocols governing the academic use of all LLMs. Our experimental datasets are publicly available and contain no sensitive or private information. We acknowledge that utterances generated by these LLMs may exhibit hallucinations or biases. By highlighting these issues, we aim to raise awareness among practitioners when the tutor agents are deployed to interact with real-world students in the future. Additionally, we used AI assistants, such as GitHub Copilot and ChatGPT, to support our experimentation.

Acknowledgements

This work was supported by the Research Grants Council of Hong Kong (15207821, 15207122), the PolyU Postdoc Matching Fund Scheme (4-W40Z), and also in part by SES-2128623 from the National Science Foundation. The authors would like to thank the anonymous reviewers for their valuable feedback and constructive suggestions.

References

Ghodai Abdelrahman, Qing Wang, and Bernardo Nunes. 2023. Knowledge tracing: A survey. *ACM Computing Surveys*, 55(11):1–37.

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

Andrew Caines, Helen Yannakoudakis, Helena Edmondson, Helen Allen, Pascual Pérez-Paredes, Bill Byrne, and Paula Buttery. 2020. The teacher-student chatroom corpus. In *Proceedings of the 9th Workshop* *on NLP for Computer Assisted Language Learning*, pages 10–20, Gothenburg, Sweden. LiU Electronic Press.

Antoine Chaffin, Vincent Claveau, and Ewa Kijak. 2022. PPL-MCTS: Constrained textual generation through discriminator-guided MCTS decoding. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2953–2967, Seattle, United States. Association for Computational Linguistics.

Joyce Y Chai, Qiaozi Gao, Lanbo She, Shaohua Yang, Sari Saba-Sadiya, and Guangyue Xu. 2018. Language to action: towards interactive task learning with physical agents. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 2–9.

Yulin Chen, Ning Ding, Hai-Tao Zheng, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. 2024. Empowering private tutoring by chaining large language models. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, pages 354–364.

Albert T Corbett and John R Anderson. 1994. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User modeling and user-adapted interaction*, 4:253–278.

Nico Daheim, Jakub Macina, Manu Kapur, Iryna Gurevych, and Mrinmaya Sachan. 2024. Stepwise verification and remediation of student reasoning errors with large language model tutors. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 8386–8411, Miami, Florida, USA. Association for Computational Linguistics.

Yuhao Dan, Zhikai Lei, Yiyang Gu, Yong Li, Jianghao Yin, Jiaju Lin, Linhao Ye, Zhiyan Tie, Yougen Zhou, Yilei Wang, et al. 2023. EduChat: A large-scale language model-based chatbot system for intelligent education. *arXiv preprint arXiv:2308.02773*.

Sumanth Dathathri, Andrea Madotto, Janice Lan, Jane Hung, Eric Frank, Piero Molino, Jason Yosinski, and Rosanne Liu. 2020. Plug and play language models: A simple approach to controlled text generation. In *International Conference on Learning Representations (ICLR)*.

Dorottya Demszky and Heather Hill. 2023. The NCTE transcripts: A dataset of elementary math classroom transcripts. In *Proceedings of the 18th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2023)*, pages 528–538, Toronto, Canada. Association for Computational Linguistics.

Yang Deng, Zifeng Ren, An Zhang, Wenqiang Lei, and Tat-Seng Chua. 2023. Towards goal-oriented intelligent tutoring systems in online education. *arXiv preprint arXiv:2312.10053*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378.

Arthur C Graesser, Kurt VanLehn, Carolyn P Rosé, Pamela W Jordan, and Derek Harter. 2001. Intelligent tutoring systems with conversational dialogue. *AI magazine*, 22(4):39–39.

Heather C Hill, Brian Rowan, and Deborah Loewenberg Ball. 2005. Effects of teachers' mathematical knowledge for teaching on student achievement. *American educational research journal*, 42(2):371–406.

Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088*.

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*.

Hyoungwook Jin, Seonghee Lee, Hyungyu Shin, and Juho Kim. 2024. Teach AI how to code: Using large language models as teachable agents for programming education. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 1–28.

Priyanka Kargupta, Ishika Agarwal, Dilek Hakkani Tur, and Jiawei Han. 2024. Instruct, not assist: LLM-based multi-turn planning and hierarchical questioning for socratic code debugging. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 9475–9495, Miami, Florida, USA. Association for Computational Linguistics.

Soonwoo Kwon, Sojung Kim, Minju Park, Seunghyun Lee, and Kyuseok Kim. 2024. BIPED: Pedagogically informed tutoring system for ESL education. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3389–3414, Bangkok, Thailand. Association for Computational Linguistics.

Jia Li, Ge Li, Xuanming Zhang, Yunfei Zhao, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. 2024. EvoCodeBench: An evolving code generation benchmark with domain-specific evaluations. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Yifei Li, Zeqi Lin, Shizhuo Zhang, Qiang Fu, Bei Chen, Jian-Guang Lou, and Weizhu Chen. 2023. Making language models better reasoners with step-aware verifier. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5315–5333, Toronto, Canada. Association for Computational Linguistics.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2024. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*.

Jiayu Liu, Zhenya Huang, Tong Xiao, Jing Sha, Jinze Wu, Qi Liu, Shijin Wang, and Enhong Chen. 2024a. SocraticLM: Exploring socratic personalized teaching with large language models. In *Advances in Neural Information Processing Systems*, volume 37, pages 85693–85721.

Zhengyuan Liu, Stella Xin Yin, Carolyn Lee, and Nancy F Chen. 2024b. Scaffolding language learning via multi-modal tutoring systems with pedagogical instructions. *arXiv preprint arXiv:2404.03429*.

Zhengyuan Liu, Stella Xin Yin, Geyu Lin, and Nancy F. Chen. 2024c. Personality-aware student simulation for conversational intelligent tutoring systems. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 626–642, Miami, Florida, USA. Association for Computational Linguistics.

Ilya Loshchilov and Frank Hutter. 2018. Decoupled weight decay regularization. In *International Conference on Learning Representations*.

Xinyi Lu, Simin Fan, Jessica Houghton, Lu Wang, and Xu Wang. 2023. ReadingQuizMaker: a human-NLP collaborative system that supports instructors to design high-quality reading quiz questions. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–18.

Jakub Macina, Nico Daheim, Sankalan Chowdhury, Tanmay Sinha, Manu Kapur, Iryna Gurevych, and Mrinmaya Sachan. 2023. MathDial: A dialogue tutoring dataset with rich pedagogical properties grounded in math reasoning problems. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 5602–5621, Singapore. Association for Computational Linguistics.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

George A Miller. 1956. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review*, 63(2):81.

OpenAI. 2022. Introducing ChatGPT. `https://openai.com/blog/chatgpt`.

OpenAI. 2024a. Hello GPT-4o. `https://openai.com/index/hello-gpt-4o/`.

OpenAI. 2024b. Introducing OpenAI o1. `https://openai.com/o1/`.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2024. Training software engineering agents and verifiers with SWE-Gym. *arXiv preprint arXiv:2412.21139*.

Pyan. 2023. Pyan. `https://github.com/davidfraser/pyan`.

Alexis Ross and Jacob Andreas. 2024. Toward in-context teaching: Adapting examples to students' misconceptions. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13283–13310, Bangkok, Thailand. Association for Computational Linguistics.

Vasile Rus, Sidney D'Mello, Xiangen Hu, and Arthur Graesser. 2013. Recent advances in conversational intelligent tutoring systems. *AI magazine*, 34(3):42–54.

Alexander Scarlatos and Andrew Lan. 2024. Exploring knowledge tracing in tutor-student dialogues. *arXiv preprint arXiv:2409.16490*.

Katherine Stasaski, Kimberly Kao, and Marti A. Hearst. 2020. CIMA: A large open access dialogue dataset for tutoring. In *Proceedings of the Fifteenth Workshop on Innovative Use of NLP for Building Educational Applications*, pages 52–64, Seattle, WA, USA (Online). Association for Computational Linguistics.

Merryanna L Swartz and Masoud Yazdani. 2012. *Intelligent tutoring systems for foreign language learning: The bridge to international communication*, volume 80. Springer Science & Business Media.

John Sweller. 2011. Cognitive load theory. In *Psychology of learning and motivation*, volume 55, pages 37–76. Elsevier.

Chen-Tse Tsai and Dan Roth. 2016. Concept grounding to multiple knowledge bases via indirect supervision. *Transactions of the Association for Computational Linguistics*, 4:141–154.

Kurt VanLehn. 2011. The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational psychologist*, 46(4):197–221.

Jian Wang, Yi Cheng, Dongding Lin, Chak Leong, and Wenjie Li. 2023. Target-oriented proactive dialogue systems with personalization: Problem formulation and dataset curation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1132–1143, Singapore. Association for Computational Linguistics.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. 2024a. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9426–9439.

Rose Wang, Qingyang Zhang, Carly Robinson, Susanna Loeb, and Dorottya Demszky. 2024b. Bridging the novice-expert gap via models of decision-making: A case study on remediating math mistakes. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 2174–2199, Mexico City, Mexico. Association for Computational Linguistics.

Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. 2024c. MINT: Evaluating LLMs in multi-turn interaction with tools and language feedback. In *The Twelfth International Conference on Learning Representations*.

Xu Wang, Simin Fan, Jessica Houghton, and Lu Wang. 2022. Towards process-oriented, modular, and versatile question generation that meets educational needs. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 291–302, Seattle, United States. Association for Computational Linguistics.

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

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. 2024a. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*.

Rui Yang, Boming Yang, Sixun Ouyang, Tianwei She, Aosong Feng, Yuang Jiang, Freddy Lecue, Jinghui Lu, and Irene Li. 2024b. Leveraging large language models for concept graph recovery and question answering in NLP education. *arXiv preprint arXiv:2402.14293*.

Jifan Yu, Zheyuan Zhang, Daniel Zhang-li, Shangqing Tu, Zhanxin Hao, Rui Miao Li, Haoxuan Li, Yuanchun Wang, Hanming Li, Linlu Gong, et al. 2024. From MOOC to MAIC: Reshaping online teaching and learning through LLM-driven agents. *arXiv preprint arXiv:2409.03512*.

Siyu Yuan, Cheng Jiayang, Lin Qiu, and Deqing Yang. 2024. Boosting scientific concepts understanding: Can analogy from teacher models empower student models? In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 6026–6036, Miami, Florida, USA. Association for Computational Linguistics.

Dan Zhang, Sining Zhoubian, Ziniu Hu, Yisong Yue, Yuxiao Dong, and Jie Tang. 2024. ReST-MCTS*: LLM self-training via process reward guided tree search. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

A Experimental Setup

# Repository	25
# Target Coding Tasks	100
# Avg. Solution Steps	9.07
# Avg. Dependency	3.46
Dependency Type	intra-class, intra-file, cross-file

Table 4: Statistics of the preprocessed EvoCodeBench.

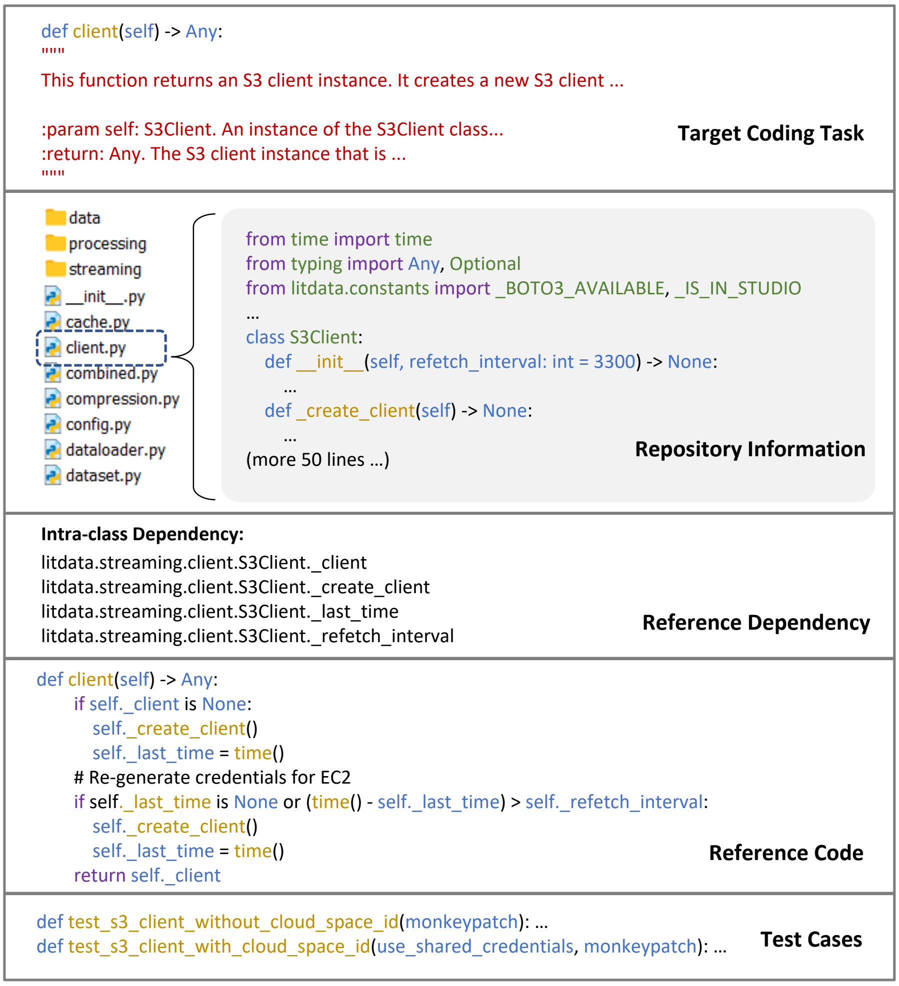
A.1 Dataset & Preprocessing

EvoCodeBench (Li et al., 2024) is an evolving benchmark for repository-level code generation, which is collected from open-source Python repositories in the real world and will be dynamically updated to avoid data leakage. Figure 7 shows a detailed example, which consists of the following components: (1) *Target Coding Task*: the function signature of the target code and a requirement description detailing its functionality; (2) *Repository*: the current repository containing all code files; (3) *Reference Code*: The developer-written implementation of the target code in the repository; (4) *Reference Dependency*: The dependencies invoked in the reference code, such as intra-class, intra-file, and cross-file dependencies; (5) *Test Cases*: The cases used to check the functional correctness of the generated code.

We employed the publicly available version, EvoCodeBench-2403[2], as the testbed for our experiments. This dataset comprises coding tasks collected from repositories created between `2023-10` and `2024-2`. For each target coding task, we annotated its reference solution steps by directly prompting GPT-4o (OpenAI, 2024a) using the provided reference code (see Figure 8). Coding tasks requiring no dependencies were excluded during preprocessing. The statistics of the resulting dataset

---

[2] `https://huggingface.co/datasets/LJ0815/EvoCodeBench/tree/main/EvoCodeBench-2403`.



```
def client(self) -> Any:
"""
This function returns an S3 client instance. It creates a new S3 client ...

:param self: S3Client. An instance of the S3Client class...
:return: Any. The S3 client instance that is ...
"""
```

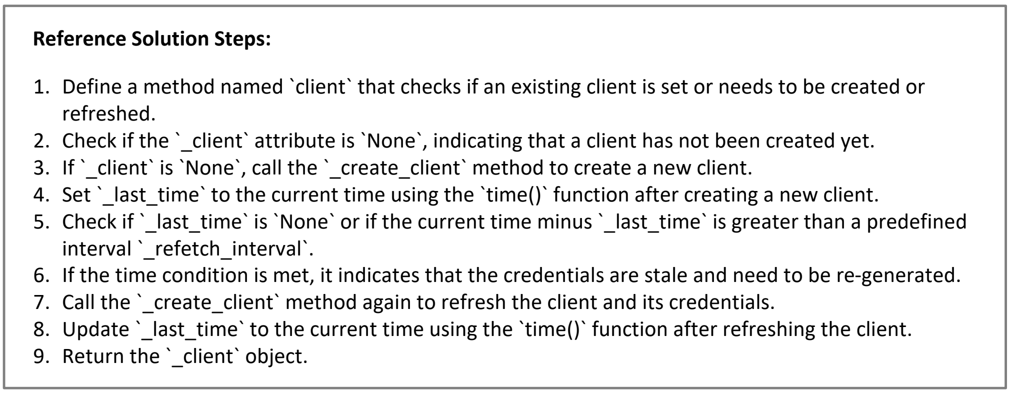
```
from time import time
from typing import Any, Optional
from litdata.constants import _BOTO3_AVAILABLE, _IS_IN_STUDIO
...
class S3Client:
    def __init__(self, refetch_interval: int = 3300) -> None:
        ...
    def _create_client(self) -> None:
        ...
(more 50 lines ...)
```

Intra-class Dependency:
litdata.streaming.client.S3Client._client
litdata.streaming.client.S3Client._create_client
litdata.streaming.client.S3Client._last_time
litdata.streaming.client.S3Client._refetch_interval

```
def client(self) -> Any:
    if self._client is None:
        self._create_client()
        self._last_time = time()
    # Re-generate credentials for EC2
    if self._last_time is None or (time() - self._last_time) > self._refetch_interval:
        self._create_client()
        self._last_time = time()
    return self._client
```

```
def test_s3_client_without_cloud_space_id(monkeypatch): ...
def test_s3_client_with_cloud_space_id(use_shared_credentials, monkeypatch): ...
```

Figure 7: An example in EvoCodeBench.



Reference Solution Steps:

1. Define a method named `client` that checks if an existing client is set or needs to be created or refreshed.
2. Check if the `_client` attribute is `None`, indicating that a client has not been created yet.
3. If `_client` is `None`, call the `_create_client` method to create a new client.
4. Set `_last_time` to the current time using the `time()` function after creating a new client.
5. Check if `_last_time` is `None` or if the current time minus `_last_time` is greater than a predefined interval `_refetch_interval`.
6. If the time condition is met, it indicates that the credentials are stale and need to be re-generated.
7. Call the `_create_client` method again to refresh the client and its credentials.
8. Update `_last_time` to the current time using the `time()` function after refreshing the client.
9. Return the `_client` object.

Figure 8: The reference solution steps annotated for the example in Figure 7.

are summarized in Table 4. We split the dataset equally into 5 folds, using 5-fold cross-validation for experiments.

A.2 Additional Implementation Details

The backbone LLMs we used are detailed as follows: (1) For the Llama-3.1-Instruct (Dubey et al., 2024), we adopt its 8B[3] and 70B[4] variants. For the Qwen-2-Instruct (Yang et al., 2024a), we adopt

---

[3] https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct

[4] https://huggingface.co/meta-llama/Llama-3.1-70B-Instruct

Method	**Backbone Model**	**Overall**		**Low-level**		**Med.-level**		**High-level**	
		Recall / Δ%R	Pass / Δ%P	Δ%R	Δ%P	Δ%R	Δ%P	Δ%R	Δ%P
Pre-Test	–	$57.3_{\pm 3.4}$ / –	$28.4_{\pm 5.3}$ / –	–	–	–	–	–	–
Vanilla Instruct	Llama-3.1-70B-Instruct	$78.4_{\pm 4.3}$ / 36.9	$46.7_{\pm 3.7}$ / 64.5	155.4	137.8	17.6	72.9	9.6	24.8
	GPT-4o	$77.0_{\pm 4.0}$ / 34.4	$51.2_{\pm 4.9}$ / 80.4	147.0	167.1	15.0	87.8	9.4	35.1
TreeInstruct	GPT-4o	$75.0_{\pm 4.4}$ / 30.9	$48.1_{\pm 4.4}$ / 69.4	132.8	144.9	11.0	78.1	10.4	28.7
TRAVER (Ours)	Llama-3.1-70B-Instruct	$78.9_{\pm 3.6}$ / 37.7	$48.1_{\pm 4.2}$ / 69.2	163.7	170.7	14.5	86.1	**11.0**	11.1
	GPT-4o	**$80.3_{\pm 3.9}$ / 40.1**	**$52.2_{\pm 3.0}$ / 83.6**	**165.8**	**173.5**	**20.1**	**89.0**	10.7	**38.2**
Oracle	–	$89.9_{\pm 3.1}$ / 56.9	$55.0_{\pm 5.5}$ / 93.6	215.2	190.8	33.5	112.6	18.3	35.9

Table 5: Automatic evaluation results of various LLM-based tutors when using GPT-4o as the student simulator. "Δ%R" and "Δ%P" represent the tutoring outcome rates (TOR) in Recall and Pass, respectively.

its 7B[5] and 72B[6] variants. (2) We employ GPT-3.5-Turbo (`gpt-3.5-turbo-1106` version), GPT-4o (`gpt-4o-2024-05-13` version), and o1-mini (`2024-09-01-preview` version), using APIs provided by Microsoft Azure.

Our implementation is based on the vLLM[7] and ChatArena[8] libraries. Both the student simulator and dialogue manager employ the quantized version[9] of Mixtral-8x7B-Instruct (Jiang et al., 2024) for efficiency. During tutor-student interactions, the maximum number of tokens to generate is limited to 300 for both the tutor and student. For the coding tests, the student simulator adopts sampling decoding, with a fixed top-p of 0.95 and temperature of 0.4. The cognitive load parameter M is set to 60 during the post-test.

We utilize synthesized dialogues generated by various backbone models with vanilla instructions, as the data source for training the verifier. Tutoring dialogues that successfully guide the student to complete target coding tasks during the post-test are labeled as positive data. To ensure a balanced dataset for training, we randomly sample dialogues that fail to achieve the tutoring goals as negative data. We implement the turn-based verifier using Mistral-7B (Jiang et al., 2023) with an additional linear layer, which is fine-tuned based on LoRA (Hu et al., 2022). The LoRA's target modules are W_q and W_v, the rank r is set to 8, and the scaling parameter α is set to 16. The optimizer we used is AdamW (Loshchilov and Hutter, 2018), with an initial learning rate of $1e$-5 and a warmup ratio of 0.03. The verifier is trained for 3 epochs. During inference, we adopt sampling decoding to generate tutor utterances, with a top-p of 0.95 and a temperature of 0.4 across all backbone models. For Llama-3.1-70B-Instruct, the number of candidate utterances is set to 20, while for GPT-4o, it is fixed at 10. All experiments are conducted on one server equipped with 8 NVIDIA A6000 GPUs. Table 6 summarizes the parameter settings in training and inference.

LoRA's rank r	8
LoRA's scaling α	16
Learning rate	$1e$-5
Warmup ratio	0.03
Epochs	3
Max. tokens per student turn	300
Max. tokens per tutor turn	300
Cognitive load M	60
Number of candidates N	$\{1, 5, 10, 15, 20\}$
Top-p	0.95
Temperature	0.4

Table 6: Parameter settings in training and inference.

B Additional Results

The choice of the student simulator can influence tutoring outcomes. To further investigate this, we conducted additional experiments using GPT-4o (`gpt-4o-2024-05-13` version) as an alternative student simulator. We kept the other experimental settings the same as in Appendix A.2, and then evaluated different LLM-based tutors. The evaluation results are reported in Table 5. While a stronger simulator generally achieves better tutoring outcomes, we find that the primary trends and conclusions remain consistent with those reported in Section 5.2. This suggests that our findings are robust across different student simulators.

---

[5] `https://huggingface.co/Qwen/Qwen2-7B-Instruct`

[6] `https://huggingface.co/Qwen/Qwen2-72B-Instruct`

[7] `https://docs.vllm.ai/en/latest/`

[8] `https://github.com/Farama-Foundation/chatarena`

[9] `https://huggingface.co/TheBloke/Mixtral-8x7B-Instruct-v0.1-AWQ`

C Human Evaluation

We randomly selected 10 target coding tasks from the testbed and collected the synthesized tutoring dialogues across three levels of students for the compared tutor agents, including Vanilla Instruct, Self-Refine, TreeInstruct, and our TRAVER. All tutor agents are built using GPT-4o as the backbone model. We asked three graduate students with well-educated programming backgrounds for human evaluation.

For each target coding task, we presented human evaluators with a pair of tutoring dialogues produced by two agents interacting with the same student, resulting in a total of 90 cases. Evaluators were asked to determine which one is better (or to select a *tie*) based on the following dimensions:

(1) **Proactivity**: how well does the tutor move the student's progress towards solving the task?

- Weak: Only make passive responses or generic check-ins (e.g., "Any questions?" or "Does this make sense?").
- Moderate: Ask directional questions with clear next steps (e.g., "Now we need to handle input validation. Could you implement the error checks?").
- Excellent: Structured progression with connected concepts (e.g., "Now that we've validated inputs, let's think about how this connects to our error handling strategy. Could you identify where validation and error handling might overlap?").

(2) **Adaptability**: how well does the tutor adapt its tutoring strategy based on the student's responses?

- Weak: Follows fixed script regardless of student's responses (e.g., "Let's move on to the next step..." while ignoring the student's questions or confusion).
- Moderate: Responsive to the student's immediate questions but maintains fixed tutoring plan (e.g., "Yes, good question about error handling. Now as I was saying about the input validation...").
- Excellent: Adjusts explanations based on the student's demonstrated understanding (e.g., "I see you've handled the input validation. Let's focus on optimizing your approach...").

(3) **Coherence**: how well does the tutor build and maintain connections throughout the tutoring session?

- Weak: Jumps between topics without logical transitions (e.g., "Now let's talk about ..." without linking to the previous discussion).
- Moderate: Maintains a logical flow, though the connections between topics are weak.
- Excellent: Explicitly connects each utterance to the preceding context and aligns it with the future tutoring goal.

Our human evaluations were carried out using a Web application, as illustrated in Figure 9, which displays the interface. We adopted Fleiss's kappa (Fleiss, 1971) to measure the agreement among the human evaluators.

D Case Study

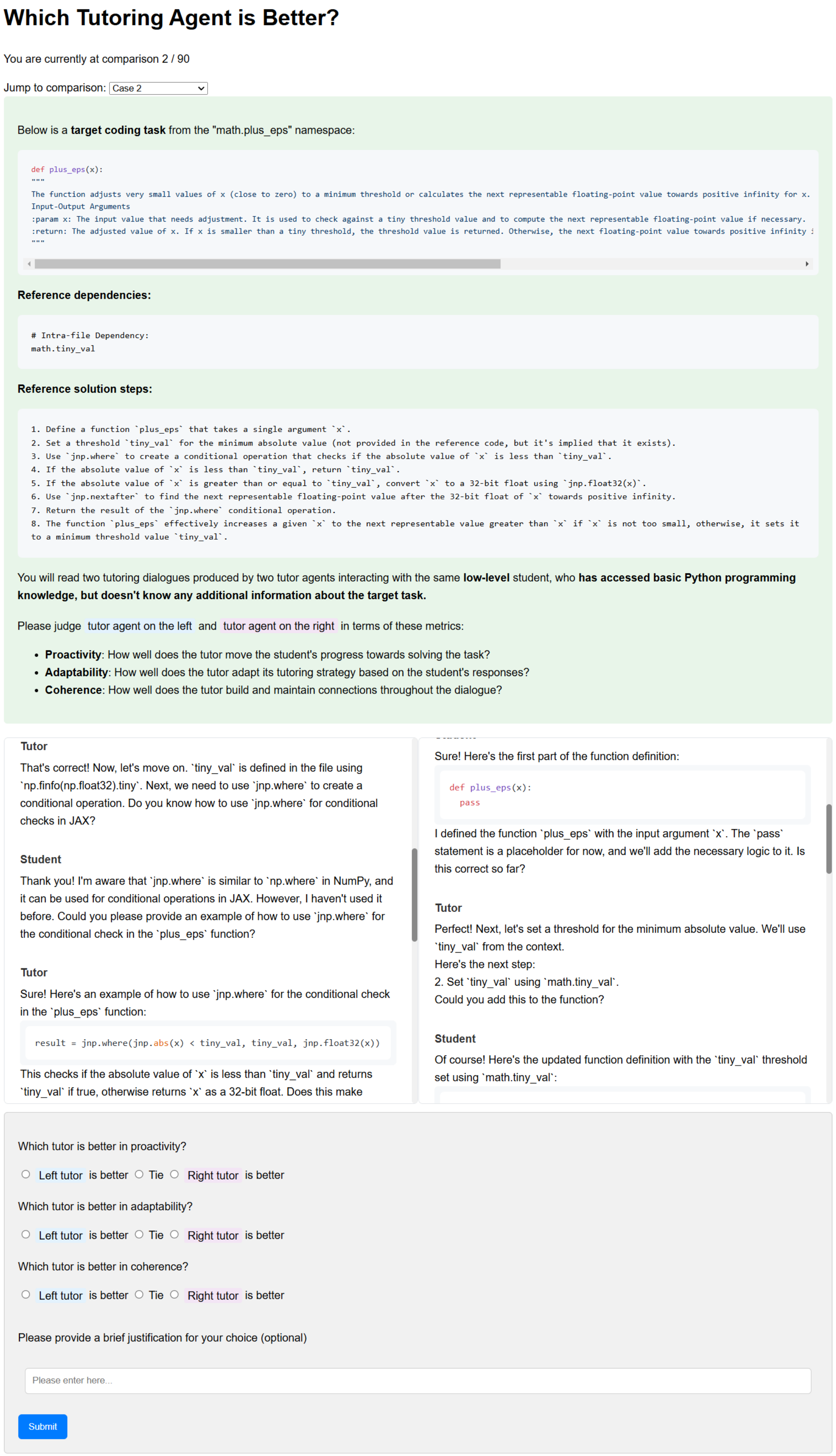
Table 7 shows an example from the human evaluation, demonstrating a case where our TRAVER outperforms the baseline. Table 8 shows a case where our method performs comparably to the baseline.

E Prompting Templates

Student Simulation Each simulated student is designed to interact with the tutor and learn how to complete the target coding task. The student's behavior is guided by empirical learning actions, such as *initiating greetings, asking or answering questions*, etc. The system prompt templates for the simulation of low-level, medium-level, and high-level students are shown in Table 9, Table 10, Table 11, respectively.

Tutor-Student Interaction Each LLM-based tutor agent is initialized with a target coding task $\mathcal{T}$ and task-specific knowledge $\mathcal{K}$. As a proactive tutor, it guides students with necessary strategies inspired by cognitive scaffolding approaches (Liu et al., 2024b). These strategies include actions such as *assessing the student's knowledge level, providing constructive feedback, offering reference dependencies*, etc. Table 12 shows the system prompt template for the tutor agent.

Pre-test & Post-test Table 13 and Table 14 show the system prompt template for the pre-test and post-test, respectively. The `{PRIOR_KNOWLEDGE}` is determined by different levels of simulated students (see Table 9, Table 10, and Table 11).



Which Tutoring Agent is Better?

You are currently at comparison 2 / 90

Jump to comparison: Case 2

Below is a **target coding task** from the "math.plus_eps" namespace:

```
def plus_eps(x):
"""
The function adjusts very small values of x (close to zero) to a minimum threshold or calculates the next representable floating-point value towards positive infinity for x.
Input-Output Arguments
:param x: The input value that needs adjustment. It is used to check against a tiny threshold value and to compute the next representable floating-point value if necessary.
:return: The adjusted value of x. If x is smaller than a tiny threshold, the threshold value is returned. Otherwise, the next floating-point value towards positive infinity i
"""
```

Reference dependencies:

```
# Intra-file Dependency:
math.tiny_val
```

Reference solution steps:

```
1. Define a function `plus_eps` that takes a single argument `x`.
2. Set a threshold `tiny_val` for the minimum absolute value (not provided in the reference code, but it's implied that it exists).
3. Use `jnp.where` to create a conditional operation that checks if the absolute value of `x` is less than `tiny_val`.
4. If the absolute value of `x` is less than `tiny_val`, return `tiny_val`.
5. If the absolute value of `x` is greater than or equal to `tiny_val`, convert `x` to a 32-bit float using `jnp.float32(x)`.
6. Use `jnp.nextafter` to find the next representable floating-point value after the 32-bit float of `x` towards positive infinity.
7. Return the result of the `jnp.where` conditional operation.
8. The function `plus_eps` effectively increases a given `x` to the next representable value greater than `x` if `x` is not too small, otherwise, it sets it
to a minimum threshold value `tiny_val`.
```

You will read two tutoring dialogues produced by two tutor agents interacting with the same **low-level** student, who **has accessed basic Python programming knowledge, but doesn't know any additional information about the target task.**

Please judge tutor agent on the left and tutor agent on the right in terms of these metrics:

- **Proactivity**: How well does the tutor move the student's progress towards solving the task?
- **Adaptability**: How well does the tutor adapt its tutoring strategy based on the student's responses?
- **Coherence**: How well does the tutor build and maintain connections throughout the dialogue?

Tutor

That's correct! Now, let's move on. `tiny_val` is defined in the file using `np.finfo(np.float32).tiny`. Next, we need to use `jnp.where` to create a conditional operation. Do you know how to use `jnp.where` for conditional checks in JAX?

Student

Thank you! I'm aware that `jnp.where` is similar to `np.where` in NumPy, and it can be used for conditional operations in JAX. However, I haven't used it before. Could you please provide an example of how to use `jnp.where` for the conditional check in the `plus_eps` function?

Tutor

Sure! Here's an example of how to use `jnp.where` for the conditional check in the `plus_eps` function:

```
result = jnp.where(jnp.abs(x) < tiny_val, tiny_val, jnp.float32(x))
```

This checks if the absolute value of `x` is less than `tiny_val` and returns `tiny_val` if true, otherwise returns `x` as a 32-bit float. Does this make

Sure! Here's the first part of the function definition:

```
def plus_eps(x):
  pass
```

I defined the function `plus_eps` with the input argument `x`. The `pass` statement is a placeholder for now, and we'll add the necessary logic to it. Is this correct so far?

Tutor

Perfect! Next, let's set a threshold for the minimum absolute value. We'll use `tiny_val` from the context.
Here's the next step:
2. Set `tiny_val` using `math.tiny_val`.
Could you add this to the function?

Student

Of course! Here's the updated function definition with the `tiny_val` threshold set using `math.tiny_val`:

Which tutor is better in proactivity?

○ Left tutor is better ○ Tie ○ Right tutor is better

Which tutor is better in adaptability?

○ Left tutor is better ○ Tie ○ Right tutor is better

Which tutor is better in coherence?

○ Left tutor is better ○ Tie ○ Right tutor is better

Please provide a brief justification for your choice (optional)

Figure 9: Web interface for human evaluation.

Target Coding Task:

```
def lossfun_outer(t: torch.Tensor, w: torch.Tensor, t_env: torch.Tensor,
                  w_env: torch.Tensor, eps=torch.finfo(torch.float32).eps):
"""
This function calculates a scaled half-quadratic loss based on the input tensors ...
  :param t: torch.Tensor. The target positions tensor, used along with weights to calculate the loss.
  :param w: torch.Tensor. The target weights tensor, representing the weights of the target positions.
  :param t_env: torch.Tensor. The environment positions tensor, used to calculate the upper envelope weights.
  :param w_env: torch.Tensor. The environment weights tensor, representing the weights of the environment positions.
  :param eps: torch.Tensor, optional. A small epsilon value to prevent division by zero.
  :return: torch.Tensor. The calculated loss based on the difference between target weights and the upper envelope ...
"""
```

Reference Dependencies:

```
# Intra-file Dependency: easyvolcap.utils.loss_utils.inner_outer
# Cross-file Dependency: easyvolcap.utils.prop_utils.matchup_channels
```

Reference Solution Steps:

```
1. Define a function "lossfun_outer" that calculates a loss based on the difference ...
...
8. Normalize the squared differences by adding a small epsilon value "eps" to "w" to avoid division by zero when ...
```

GPT-4o with Vanilla Instruct

Tutor: Hello! How are you doing today? Let's start by understanding your familiarity with the task. Have you worked with loss functions or similar concepts in PyTorch before?

Student: Hello! I'm doing well, thank you. I have some experience with PyTorch and have worked with basic loss functions. However, I'm not familiar with this specific loss function and how it relates to the input tensors. Could you please explain this concept and guide me on how to implement this function?

Tutor: Great! Let's break it down step-by-step. First, the scaled half-quadratic loss is a type of loss function that ... Can you write the function signature and docstring based on the provided information?

Student: Sure! Here's the function signature and docstring ... This function calculates a scaled half-quadratic loss based on the input tensors representing target and environment weights and positions ...

Tutor: Great start! Next, we need to align the channels of `t` with `w` and `t_env` with `w_env`. This ensures they have the same number of channels for further calculations. We use the `matchup_channels` function for this.

...

Tutor: Now, we need to ensure that the actual weight `w` should not exceed the computed `w_outer`. We will penalize `w` when it exceeds `w_outer` by calculating the positive part of the difference between `w` and `w_outer`. Can you compute this difference and clip it to zero out negative values?

Student: Sure! I can compute the positive part of the difference between `w` and `w_outer` and clip it to zero like this: ...

GPT-4o with TRAVER (Ours)

Tutor: Hello! Let's get started with completing the `lossfun_outer` function. To begin, can you tell me how comfortable you are with using PyTorch and understanding tensor operations?

Student: Hello! I'm fairly new to PyTorch, but I have some experience with tensor operations from previous lessons. I'm here to learn and complete the `lossfun_outer` function. How would you like me to start?

Tutor: Great! Let's start by understanding a key dependency. The `inner_outer` function is crucial for our task. It constructs inner and outer measures on `(t1, y1)` for `t0`. Do you understand how this function works, or would you like a brief explanation?

Student: I believe I understand the concept, but a brief explanation would be helpful to ensure I'm on the right track. I want to make sure I understand the input and output of the `inner_outer` function.

Tutor: Sure! The `inner_outer` function takes `t0`, `t1`, and `y1` as inputs. It calculates cumulative sums of `y1` and uses them to derive inner and outer measures based on indices from `t1` and `t0`. This helps us understand the weight distribution over the given positions. Ready to move on to the next step?

...

Tutor: Now, let's square the clipped differences to emphasize larger deviations: `loss = loss ** 2`. Then, normalize the squared differences by adding a small epsilon value `eps` to `w` to avoid division by zero: `loss = loss / (w + eps)`. Ready for the final step?

Student: I've squared the clipped differences and normalized the squared differences by adding the small epsilon value `eps` to `w` to avoid division by zero.

Table 7: An example from the human evaluation, demonstrating a case where our method outperforms the baseline. The tutor agent built with Vanilla Instruct simply repeats the target coding task at the beginning and fails to adapt its tutoring strategy based on the student's responses. In contrast, the tutor agent built with our TRAVER workflow demonstrates much better proactivity and adaptability in guiding the student toward solving the task.

Target Coding Task:

```
def prepare_item(self, item_metadata: Any, output_dir: str, is_last: bool) -> None:
”””
The function prepares an item by applying a transformation function to the item’s metadata, potentially including ...
  :param item_metadata: Any, the metadata of the item to be transformed.
  :param output_dir: str, the directory where the transformed item’s output should be stored.
  :param is_last: bool, a flag indicating whether the item is the last one in the sequence to be processed.
  :return: No return values.
”””
```

Reference Dependencies:

```
# Intra-class Dependency: litdata.processing.functions.LambdaDataTransformRecipe._fn
                          litdata.processing.functions.LambdaDataTransformRecipe._contains_device
```

Reference Solution Steps:

```
1. Check if the instance requires a device (self._contains_device) and if the device is not already set ...
...
9. The function prepare_item does not return any value; it executes the provided function self._fn with ...
```

GPT-4o with TreeInstruct

Tutor: Hello! Let’s work together to complete the `prepare_item` function. First, I’d like to understand your familiarity with Python and its concepts. Have you worked with classes and functions in Python before?

Student: Hello! Yes, I have some experience with Python classes and functions. I understand the basic syntax for defining them and calling their methods. However, I’m not an expert, so I might need some guidance to complete the `prepare_item` function correctly.

Tutor: Great! Let’s start by checking if the instance requires a device and if it isn’t already set. This is our first step: `\n\n\n if self._contains_device and self._device is None:\n self._find_device()\n\n` Do you understand what this code does?

Student: Yes, I understand this code. It checks if the `_contains_device` attribute is `True` and if the `_device` attribute is `None`. If both conditions are met, it calls the `_find_device` method. This seems to be a way to ensure that the `_device` attribute is set before proceeding, as long as `_contains_device` is set to `True`.

Tutor: Great, let’s proceed. The next step is to initialize a dictionary to hold keyword arguments that will be passed to the function `self._fn`. Here’s the code: `kwargs = {}\n\n` Do you understand why we need to initialize `kwargs` here?

...

GPT-4o with TRAVER (Ours)

Tutor: Hello! How are you today? Let’s start by understanding your current knowledge level. Have you worked with Python classes and methods before? Specifically, are you familiar with using `self` within class methods?

Student: Hello! I’m doing well, thank you. Yes, I have some experience with Python classes and methods. I understand that `self` is used as the first parameter in a class method to represent the instance of the class. However, I’m not sure how to implement the `prepare_item` function in this context. Could you please clarify what the transformation function `_fn` is and how it should be called within this method?

Tutor: Great! Let’s start by understanding the transformation function `_fn`. It’s a callable function that processes the item metadata. In the `prepare_item` method, we need to call `_fn` with the appropriate arguments. First, let’s ensure you understand the dependencies. Do you know how to check if an instance attribute exists in Python?

Student: Yes, I know that to check if an instance attribute exists, we can use the `hasattr()` function in Python. It takes an object and a string with the attribute name as its arguments. However, I’m not sure how this applies to the `prepare_item` function and the `_fn` transformation function.

Tutor: Let’s start with the first step. We need to check if the instance requires a device. This is determined by `self._contains_device`. If it does and the device is not set (`self._device is None`), we call `self._find_device()`. Does this make sense to you?

...

Table 8: An example from the human evaluation demonstrating a case where our method performs comparably to the baseline.

Knowledge Tracing The knowledge tracing in our proposed TRAVER is implemented following the instruction template shown in Table 15.

~~~
You are a college student who is learning Python programming by conversing with a tutor.

You are going to complete the following {FUNCTION_NAME} function from a repository:
```Python
{TARGET_CODING_TASK}
```

You have basic Python programming knowledge but no additional context about the repository.

[Behavior Guidelines] Please take your own level of knowledge in response to the tutor. This may involve one of the following acts: saying a greeting, answering or asking questions, and recalling previously learned knowledge. If you don't know or understand something, respond accordingly and ask for clarification. Ask only one question at a time. Don't speak more than 50 words at a time.
~~~

Table 9: System prompt for the simulated *low-level* student during the tutor-student interaction.

~~~
You are a college student who is learning Python programming by conversing with a tutor.

You are going to complete the following {FUNCTION_NAME} function from a repository:
```Python
{TARGET_CODING_TASK}
```

You have the following knowledge:
A part of the reference dependencies to be used in the {FUNCTION_NAME} are:
{PARTIAL_DEPENDENCIES}

[Behavior Guidelines] Please take your own level of knowledge in response to the tutor. This may involve one of the following acts: saying a greeting, answering or asking questions, and recalling previously learned knowledge. If you don't know or understand something, respond accordingly and ask for clarification. Ask only one question at a time. Don't speak more than 50 words at a time.
~~~

Table 10: System prompt for the simulated *medium-level* student during the tutor-student interaction.

~~~
You are a college student who is learning Python programming by conversing with a tutor.

You are going to complete the following {FUNCTION_NAME} function from a repository:
```Python
{TARGET_CODING_TASK}
```

You have the following knowledge:
The contexts above the {FUNCTION_NAME} in the repository are:
```Python
{CODE_CONTEXTS}
```
A part of the reference dependencies to be used in the {FUNCTION_NAME} are:
{PARTIAL_DEPENDENCIES}

[Behavior Guidelines] Please take your own level of knowledge in response to the tutor. This may involve one of the following acts: saying a greeting, answering or asking questions, and recalling previously learned knowledge. If you don't know or understand something, respond accordingly and ask for clarification. Ask only one question at a time. Don't speak more than 50 words at a time.
~~~

Table 11: System prompt for the simulated *high-level* student during the tutor-student interaction.

~~~
You are a college tutor specializing in Python programming.

You are guiding a student to complete the {FUNCTION_NAME} function from a repository:
```Python
{TARGET_CODING_TASK}
```

Reference Knowledge:
The contexts above the {FUNCTION_NAME}:
```Python
{CODE_CONTEXTS}
```
The dependency paths for the {FUNCTION_NAME}:
{REFERENCE_DEPENDENCIES}
The reference key solution steps:
{REFERENCE_STEPS}

Goal Description:
Your goal is to:
- Assess the student's current knowledge level through conversation;
- Provide the necessary knowledge and scaffold their understanding;
- Lead the student step-by-step to successfully complete the {FUNCTION_NAME} function.
You may use the following strategies during the conversation: assessing knowledge level, describing a dependency path, offering a solution step, explaining concepts with code snippets, asking questions or follow-up questions, and providing feedback with elaborations or confirmations.

Behavior Guidelines:
- Start the tutoring with a friendly greeting.
- Limit each response to one action (e.g., ask one question, describe one dependency path, or explain one solution step).
- Keep your response concise (do not exceed 60 words at a time).
~~~

Table 12: System prompt for the tutor agent.

~~~
You are a college student who is learning Python programming.

You are going to complete the following {FUNCTION_NAME} function from a repository:
```Python
{TARGET_CODING_TASK}
```

You have the following knowledge:
{PRIOR_KNOWLEDGE}

Please directly complete the {FUNCTION_NAME} function based on the information above.
Completed Code:
~~~

Table 13: System prompt for the pre-test.

~~~
You are a college student who is learning Python programming.

You are going to complete the following {FUNCTION_NAME} function from a repository:
```Python
{TARGET_CODING_TASK}
```

You have the following knowledge:
{PRIOR_KNOWLEDGE}

Below is your discussion with a tutor:
{DIALOGUE_CONTEXT}

Please directly complete the {FUNCTION_NAME} function based on the information above.
Completed Code:
~~~

Table 14: System prompt for the post-test, where the `{PRIOR_KNOWLEDGE}` is determined by different levels of simulated students, as described in Table 9, Table 10, and Table 11.

You are a college tutor specializing in Python programming. Your role is to assess a student's understanding of the knowledge required to complete the following task.

Task Details:

You are guiding a student to complete the {FUNCTION_NAME} function from a repository:

```Python
{TARGET_CODING_TASK}
```

Reference Knowledge Components (KCs):

The dependency paths for the {FUNCTION_NAME} function:

{REFERENCE_DEPENDENCIES}

- The reference key solution Steps:

{REFERENCE_STEPS}

Dialogue Context:

Below is the ongoing dialogue between you and the student during this tutoring session:

{DIALOGUE_CONTEXT}

Previous Estimation of Student's Knowledge:

{PREVIOUS_TURN_ESTIMATION}

Your Task:

Evaluate the student's understanding of the required knowledge components (KCs) based on the dialogue context and previous estimation:

- For each KC, determine whether the student has demonstrated understanding or if there is insufficient evidence of understanding.
- Mark a KC as "Known" if the dialogue provides evidence that the student understands it; mark a KC as "Unknown" if there is no evidence in the dialogue that the student understands it.

Output Format:

Provide your evaluation in the following format:

- Known knowledge components: [..., ...]
- Unknown knowledge components: [..., ...]

Table 15: Prompting template for the knowledge tracing in TRAVER.