TreeSearch at SemEval-2025 Task 8: Monte Carlo Tree Search for Question-Answering over Tabular Data

Aakarsh Nair and Huixin Yang
Seminar für Sprachwissenschaft
Eberhard Karls Universität Tübingen, Germany
{aakarsh.nair, huixin.yang}@student.uni-tuebingen.de

Abstract

Large Language Models (LLMs) can answer diverse questions but often generate factually incorrect responses. SemEval-2025 Task 8 focuses on table-based question-answering, providing 65 real-world tabular datasets and 1,300 questions that require precise filtering and summarization of underlying tables.

We approach this problem as a neuro-symbolic code generation task, translating natural language queries into executable Python code to ensure contextually relevant and factually accurate answers. We formulate LLM decoding as a Markov Decision Process, enabling Monte Carlo Tree Search (MCTS) as a lookahead-based planning algorithm while decoding from the underlying code-generating LLM, instead of standard beam-search.

Execution success on synthetic tests and real datasets serves as a reward signal, allowing MCTS to explore multiple code-generation paths, validate outcomes, assign value to partial solutions, and refine code iteratively rather than merely maximizing sequence likelihood in a single step. Our approach improves accuracy by 2.38x compared to standard decoding.[1]

1 Introduction

Transformer-based LLMs (Vaswani et al., 2023) excel at question answering (QA) (OpenAI, 2024; Aaron Grattafiori, 2024) but frequently generate plausible yet factually incorrect responses (hallucinations) (Ji et al., 2023). Mitigating these errors is crucial for applications requiring precise, structured answers (Farquhar et al., 2024).

One domain where factual consistency is critical is table-based QA, where answers must be derived from structured tabular data. SemEval-2025 Task 8 (Osés Grijalba et al., 2024) provides a benchmark for evaluating LLMs in this setting, requiring models to generate accurate answers from real-world datasets. Table 1 lists sample questions from the SemEval-2025 Task 8 training set of questions along with their expected answer types.

Type	Dataset ID	Example Question
Boolean	072_Admissions	Is there an applicant with a chance above 95 per cent of getting into the university they applied to ?
Category	068_WorldBank_Awards	Which region has the most contracts?
Number	075_Mortality	What is the total sum of all death rate values ?
List[Category]	080_Books	List the categories of the first five books.
List[Number]	078_Fires	What are the 3 hottest temperatures recorded?

Table 1: Sample questions from the Semeval Task-8 training set along with their expected answer types.[2]

While databases have long relied on structured query languages (SQL) for precise data retrieval (Codd, 1970), translating natural language questions into executable queries remains a challenging task (Nan et al., 2022; Zhong et al., 2017).

In this task, we aim to leverage code-generating Large Language Models (Rozière et al., 2024) to query tabular data. That is, given a natural language query and the underlying table schema, we attempt to generate a concise Pandas Python code (Wes McKinney, 2010) to answer questions regarding the given table dataframe. This allows us to leverage existing code generation models like CodeLlama-7b-Python [3] model, which are extensively trained on Python code generation.

For reference, the organizers provide training data consisting of 1.3k data points, each of which includes a natural language question, the target answer, its data type, and relevant columns in the dataframe useful for answering the query. (Grijalba et al., 2024). Queries are run in either *lite* mode using the first 20 rows of the table or full mode using the entire table, with target answers for both modes provided. The organizers additionally provide a baseline decoder-only code LLM, based on Stable Code 3b (Pinnaparaju et al., 2024), which tries to generate Python-Pandas code to be run against the

[1] Our code is available at: https://github.com/HuixinYang/SemEval25-Task8-Adrianna-Aakarsh-Yang

[2] Full training dataset is available here.

[3] CodeLlama-7b-Python is openly available at: https://huggingface.co/codellama/CodeLlama-7b-Python-hf

underlying dataframe. Using a simple beam-search-based decoder, the LLM can achieve an accuracy of 27% in lite querying mode and 26% on full table querying mode on the task's test set.

Standard beam-search decoders maximize sequence likelihood but ignore program quality for planning. We therefore propose an MCTS-based planning decoder (Zhang et al., 2023) for CodeLlama-7b-Python, grounding table-QA in the runtime behavior of generated queries.

During the decoding process, the MCTS Planner can take advantage of a reward signal from terminal states to backpropagate and value intermediate decodings of the underlying Transformer model. The MCTS planner can balance exploration and exploitation in the token decoding process to generate a higher-quality set of possible Python-Pandas completions, running them against the dataframe and automatically generated test-cases to weigh possible next tokens through a look-ahead process, while the transformer's next token probabilities serve as a good heuristic to constrain the planner's search space. Moreover, the flexibility of the reward function allows us to use a variety of much larger models to inform our judgments of the generated program solutions. For example, we can use automatically generated test-cases from a `Qwen2.5-Coder-32B-Instruct` (Hui et al., 2024) model on synthetic data following the underlying table schema to validate and generate a reward signal for our decoding process.

2 Related Work

Reinforcement Learning (RL) planning, particularly Monte Carlo Tree Search (MCTS) (Sutton et al., 1998; Silver et al., 2016), has shown success in complex domains such as games and SQL generation from natural language (Zhong et al., 2017).

While LLMs exhibit inherent reasoning capabilities (OpenAI, 2024), incorporating explicit planning into their generation process is an active research area. Approaches range from prompting strategies like Chain-of-Thought (Wei et al., 2023) to models with dedicated search, reward, and reasoning components (Hao et al., 2024).

3 Our Approach

Our approach applies planning-based transformer decoding (Zhang et al., 2023) to table-question answering. Novel query program code synthesis is formulated as a Markov Decision Process (Sutton et al., 1998). A partial program along with its prompting description is considered to be a *state* s. The act of selecting a next-token from the underlying Transformer vocabulary is considered an *action* a. Thus *transition function* moves from one partial program to another by concatenating a selected token to one partial program to form another until a terminal token is appended. The *reward* for a program is a function of the validity of the program, along with the number of synthetic test-cases the generated program passes. The aim is to use the LLM to search for a path which maximizes expected future reward: $\sum_{t=0}^{n} r(s_t, a_t)$.

3.1 Monte-Carlo Tree Search Decoding

A *Monte-Carlo Tree Search (MCTS)*-based planner maximizes the accumulated reward of the generated program, replacing beam-search, which prioritizes similarity to reference solutions but cannot explicitly optimize execution quality, making it sample-inefficient.

MCTS (Silver et al., 2016) treats planning as a look-ahead search through a tree of *actions* to find a path to terminal nodes with the highest rewards. Search proceeds from the root node, with child nodes representing next-token actions selected by the planning algorithm in phases meant to balance *exploration* and *exploitation*. The four phases are: *Selection*: A process of selecting which of the root's child node to examine, *Expansion*: A process of adding child nodes for the top-k most likely next tokens for given node (as guided by the Transformer model), *Evaluation*: Expanding greedily using beam-search on Transformer model to the terminal node to estimate program reward and *Backpropagation*: Updating a node's ancestor's with visit counts and ground truth observed rewards. Throughout its execution for each node, the planning algorithm maintains a node-visit count and $Q(s, a)$, which is the average reward for the algorithm taking action a when starting from state s. A *rollout* consists of a one full generation of a sample program and its final computed *reward*. All four phases are run as part of every full rollout. The algorithm maintains a search tree of tokens till the required number of rollouts are processed.

Selection: This process starts at the root node and recursively selects its child node until a leaf node is reached. The root node consists of the full prompt up until the slotted location for query generation. Its children represent the possible next tokens in the generation. The selection phase recursively traverses and selects actions a down till it reaches an unexpanded node using a variant of the Predictive Upper Confidence Bound(P-UCB) (Silver et al., 2016; Zhang et al., 2023) action selection criterion, which balances *exploration* with *exploitation*, when selecting next token a to explore.

$$\text{argmax}_a \text{P-UCB}(s,a) = Q(s,a) + C \cdot P_{\text{LLM}}(a|s) \cdot \frac{\sqrt{\ln N(s)}}{1+N(s')}$$

Where $Q(s,a)$ is the reward of the best program generated from this node, $P_{\text{LLM}}(a|s)$ is the transformer's predicted probability that a is the next token and $N(s)$ and $N(s')$ are the prior visit counts of states s and the state s' achieved when we transition from state s to state s' by taking the token-concatenation action a, C is a ucb-constant which is used to control the scaling of the *exploration* term.

Expansion: Once we reach an unexpanded/unexplored node, we discover its possible succeeding children by using the Code LLM *Top-k* next tokens. Thus, the language model constrains the search paths of next tokens, reducing the probability that we will sample a syntactically invalid next token. The k represents the maximum child count of a node. It determines the fan-out size of our tree. Nodes for each of the sampled next tokens are created and added to the search tree.

Evaluation/Simulation: As the node added to the tree may be a partial program, LLM *beam-search* is used to generate a possible full completion of the program till a terminal node. The full-suite test-cases are run on this greedily generated program, if the program generated by this default policy is executable, the observed reward is recorded along with the rollout.

Success in compiling and executing the program against the underlying dataframe and the number of test-cases passed on a synthetic dataframe contributes to the reward for this rollout.

Backpropagation: In the final phase, the observed reward and visitation count are populated up through the ancestors of the current node to contribute to future look-ahead searches and the ancestor's P-UCB criterion, leading to refining the tree exploration policy over each subsequent rollout.

Answer Selection: The final results of running all rollouts are a dictionary of all programs generated and their corresponding observed rewards from the evaluation phases. The chosen program is one that achieves the maximum rewards, with the majority computed answer used to break tied rewards.

4 Experimental Setup

4.1 MCTS Decoder Configuration

We run the MCTS decoder (Zhang et al., 2023) for the base model `CodeLlama-7b-Python-hf` (Rozière et al., 2024), using a *horizon* of 32, which controls the maximum number of steps taken or tokens produced. 100 *rollouts* are performed, determining the number of programs generated for each question in the test set.

We use *P-UCB* as the *node selection algorithm* during the selection phase, with an expansion *width* of 5, which specifies the number of children each node can expand into. No *temporal discounting* is applied ($\gamma = 1$), meaning rewards are not penalized based on the number of steps taken to achieve them.

The base model is sampled using $\text{top}_k = 3$ and $\text{top}_p = 0.9$, with a *temperature* of 0.2 during the simulation/evaluation phase. Extensive hyperparameter tuning will be considered in future work on this task.

4.2 Enriched Prompting

We enrich the root prompt used by the base decoder to contain contextually relevant information that might be helpful during the MCTS decoding. Thus, our root prompt contains: (1) Entire dataframe *schema* for the table, we are generating our prompt, including column names and types (2) A *predicted return type* depending on the question we are answering. We recognize 5 output categories for the task. *boolean*, *category*, *number*, *list[category]* (for lists of categories), and *list[number]* (for lists of numbers). (3) *predicted columns used*, our prediction of the columns most relevant for answering the question.

We use the open source code LLM `Llama-3.1-Nemotron-70B-Instruct` (Wang et al., 2024), to predict the question category, and `Qwen2.5-Coder-32B-Instruct` (Hui et al., 2024) to predict the columns to be used for answering user's natural language query.

4.3 Reward Function

The MCTS function is highly sensitive to the design of the reward function, as the presence or absence of rewards guides the tree search procedure. As user questions are open-ended, it can often be hard to verify whether the generated code and responses are indeed accurate.

We use a three-pronged approach to guessing at the user's intent. (1) First, we use a strictly larger model to predict from the user's query the most probable output type. This probable output type is used to inform both the prompt used during the decoding process, as well as *type-check* the generated responses. (2) Additionally, the generated code is evaluated using a Python interpreter with the dataframe in question in the context of the interpreter. The executable section of the code is *parsed* and extracted from the generated model response. (3) Thirdly, the executable code fragment is run against a synthetic *test-suite* to verify that it passes multiple tests.

Sanity Type Checking and Malformed Code Detection We extract the completion's Pandas query as a single line following the prompt's return statement, truncating any additional extraneous tokens. While currently no explicit token penalty is applied, the short *horizon* biases the model toward concise outputs. The extracted code is then executed against the relevant dataframe, which is added to the interpreter context, and if a runtime error occurs *or* the resulting semantic data type differs from the expected output type, a single error penalty of -1 is imposed. This combination of execution-based feedback and type validation ensures that completions align with each query's requirements.

Test Set Generation In the spirit of *test-driven development*, we leverage large open-source models (`Llama-3.1-Nemotron-70B-Instruct`, `Qwen2.5-Coder-32B-Instruct`) to generate test-cases for a given query. Specifically, we prompt these models with the table schema and instruct them to generate randomized data, parameterized by a *random seed*, while ensuring compliance with the dataframe schema. Since these models have access to the table schema, sample data, and the user's query, they can produce meaningful assertions for validating Python Pandas queries.

During the *evaluation phase*, we run the MCTS- decoder's generated Pandas on the dummy

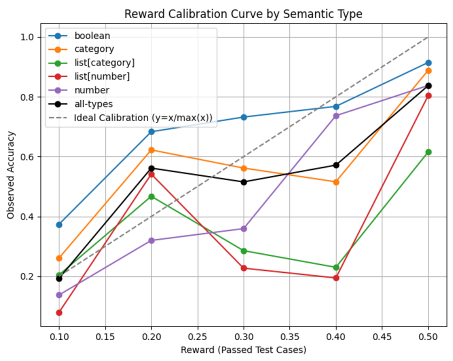

Figure 1: Plot shows the reward calibration for rollouts that passed at least one of the test-cases. We note that boolean, category, and number show good calibration, while list[category], list[number] show relatively poorer calibration, indicating that further testing is required during rollout computation to ensure their accuracy. A maximum of 5 tests are run for each rollout with a random seed. With a reward of 0.1 for each successfully passed test.

dataframe and verify that all assertions hold. Using different random seeds allows us to execute the query multiple times with varying dummy data, increasing robustness. Additionally, leveraging a diverse ensemble of models for test-case generation enhances confidence in the correctness of a query that satisfies multiple test-cases, rather than relying on any single test instance. A total of five tests are run for each query, with each passing test contributing 0.1 to the final reward value. [4]

Reward Value We use a mixture of penalties for faulty generation and rewards based on the number of tests passed to evaluate MCTS rollouts.

$$r(\text{Query}) = -1 \cdot \mathbb{I}_{\text{error}} + 0.1 \cdot p \cdot (1 - \mathbb{I}_{\text{error}})$$

Where p is the number of test-cases passed by the completion, and $\mathbb{I}_{\text{error}}$ is binary indicator function with values 0 or 1 where 1 indicates the program encounters and error during execution.

5 Results

Table 2 shows the results of running the MCTS decoder with our reward function; we also compare these results by output type category. We note that the use of a reward function leads to substantial

[4] Set of generated tests for training set for competition are available at: `https://huggingface.co/datasets/aakarsh-nair/semeval-2025-task-8-test-cases-competition`

Approach / Output Type Detail	Base Accuracy (%)	Lite Accuracy (%)
Stable Code-3b-GGUF + Beam Search (Baseline Overall)	26 %	27 %
CodeLlama-7b Python + MCTS Decoding (Our Approach)		
Overall	**61.68**	**64.36**
Breakdown by Output Type:		
Boolean	**76.74**	**74.41**
Category	64.86	67.57
Number	62.82	66.02
List [Category]	50.00	52.77
List [Number]	45.05	53.84

Table 2: Overall numerical accuracy comparison between our MCTS-based approach (CodeLlama-7b Python + MCTS decoding) and the baseline (Stable Code-3b-GGUF + Beam Search). The table also provides a detailed breakdown of the MCTS approach's accuracy by output type. Our method significantly outperforms the baseline, with boolean questions yielding the highest accuracy.

improvement over the baseline. Figure 1 shows that for atomic return types such as *boolean*, *category*, and *number*, rewards are well calibrated. That is, passing a higher number of tests in the synthetic test suite corresponds to higher observed accuracy on the evaluation benchmark.

As the MCTS decoder does not rely on in-context learning, accuracy for both lite and base strategies is roughly equivalent. We note that the results requiring list outputs tend to have the worst performance. While boolean outputs have the highest accuracy level. Table 2 we note that MCTS decoding has a baseline accuracy of 61.68% on base tests and 64.36% on lite tests, compared to the baseline provided by the host on the test set which is 27% base and 26% on the lite dataset, corresponding to a relative improvement of 128.44% on the base test set and 147.54% on the lite test set compared to the baseline.

Output Type	Category Prediction (%)
Boolean	**100.00**
Category	**100.00**
Number	96.79
List [Category]	98.61
List [Number]	82.41
Overall	**95.78**

Table 3: Category prediction accuracy by output type, used in the reward signal. Boolean and Category types showed 100% prediction accuracy.

6 Conclusion

We applied an MCTS-based decoder (Zhang et al., 2023) for code generation in SemEval-2025 Task 8 table-QA.

Unlike conventional autoregressive methods that rely on greedy decoding or single-step chain-of-thought, our approach generates multiple candidate programs and refines them through look-ahead planning. Experimental results show that MCTS decoding achieves a substantial accuracy improvement (61.68% vs. 26% baseline decoding), demonstrating the effectiveness of search-based reasoning for code generation tasks.

Our techniques leverage strong open-source models to guide a smaller local model's decoding, enabling diverse solution generation. Tree search with partial reward signals (e.g., passed tests, semantic type checks) refined solutions by balancing exploration and exploitation. Experiments also revealed that list output types require more robust checks than atomic types, where rewards were better calibrated to accuracy.

This work highlights tree-based planning's potential in code generation for table-centric QA, suggesting avenues for advanced reasoning techniques.

7 Limitation and Future Work

While our MCTS-based approach for table-centric QA significantly boosts accuracy, several limitations remain. First, we frequently observe *semantically identical but syntactically distinct* programs, resulting in unnecessary redundancy. This is compounded by occasional *extraneous tokens* at the end of generated completions, which we trim to prevent run-time errors but consequently reduce program diversity. Developing more robust code filters or pruning heuristics could improve the uniqueness and readability of generated solutions.

Second, although MCTS captures partial credit via our reward function, *reward design* remains imperfect. For example, incomplete or slightly incorrect solutions may pass partial tests without reflecting deeper logical errors. Future work could explore more fine-grained reward signals, such as dynamic coverage metrics or adversarial test generation, to improve the fidelity of feedback.

Third, we currently *do not fine-tune the model* on the MCTS rollouts or incorporate reward signals into a specialized training loop. Integrating *Expert Iteration* (Anthony et al., 2017) or iterative feedback mechanisms could further refine the policy beyond what standard prompting achieves. Additionally, we have not conducted extensive *hyperparameter searches* for aspects such as number of rollouts or maximum program length, potentially leaving performance gains on the table. These ablations and their performance tradeoffs will be explored in future works.

Lastly, our experiments focus on single-table question answering. Many real-world tasks involve *multiple tables* and heterogeneous data sources, requiring more advanced data integration strategies. Future iterations of our system could merge or join multiple dataframes, broadening its applicability to multi-step queries.

8 Acknowledgments

We would like to thank Dr. Çagrı Çöltekin for his guidance, feedback, and boundless optimism, which helped in the completion of this project. The authors also acknowledge support by the state of Baden-Württemberg through the bwHPC cluster on which most of the experiments were run.

References

Abhinav Jauhri et al. Aaron Grattafiori, Abhimanyu Dubey. 2024. The llama 3 herd of models.

Thomas Anthony, Zheng Tian, and David Barber. 2017. Thinking fast and slow with deep learning and tree search.

Edgar F Codd. 1970. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387.

Sebastian Farquhar, Jannik Kossen, Lorenz Kuhn, and Yarin Gal. 2024. Detecting hallucinations in large language models using semantic entropy. *Nature*, 630(8017):625–630.

Jorge Osés Grijalba, Luis Alfonso Ureña-López, Eugenio Martínez Cámara, and Jose Camacho-Collados. 2024. Question answering over tabular data with databench: A large-scale empirical evaluation of llms. In *Proceedings of LREC-COLING 2024*, Turin, Italy.

Shibo Hao, Yi Gu, Haotian Luo, Tianyang Liu, Xiyan Shao, Xinyuan Wang, Shuhua Xie, Haodi Ma, Adithya Samavedhi, Qiyue Gao, Zhen Wang, and Zhiting Hu. 2024. Llm reasoners: New evaluation, library, and analysis of step-by-step reasoning with large language models.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report.

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *ACM computing surveys*, 55(12):1–38.

L. Nan et al. 2022. Fetaqa: Free-form table question answering. *Transactions of the Association for Computational Linguistics*, 10:35–49.

OpenAI. 2024. Gpt-4 technical report.

Jorge Osés Grijalba, L. Alfonso Ureña-López, Eugenio Martínez Cámara, and Jose Camacho-Collados. 2024. Question answering over tabular data with DataBench: A large-scale empirical evaluation of LLMs. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 13471–13488, Torino, Italia. ELRA and ICCL.

Nikhil Pinnaparaju, Reshinth Adithyan, Duy Phung, Jonathan Tow, James Baicoianu, Ashish Datta, Maksym Zhuravinskyi, Dakota Mahan, Marco Bellagente, Carlos Riquelme, and Nathan Cooper. 2024. Stable code technical report.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489.

Richard S Sutton, Andrew G Barto, et al. 1998. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention is all you need.

Zhilin Wang, Alexander Bukharin, Olivier Delalleau, Daniel Egert, Gerald Shen, Jiaqi Zeng, Oleksii Kuchaiev, and Yi Dong. 2024. Helpsteer2-preference: Complementing ratings with preferences.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models.

Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, pages 56 – 61.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. 2023. Planning with large language models for code generation.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning.