TreeRanker: Fast and Model-agnostic Ranking System for Code Suggestions in IDEs

Daniele Cipollone[1,2], Egor Bogomolov[1], Arie van Deursen[2], Maliheh Izadi[2]

[1]JetBrains, Amsterdam, The Netherlands
[2]Delft University of Technology, Delft, The Netherlands

***Abstract*—Token-level code completion is one of the most critical features in modern Integrated Development Environments (IDEs). It assists developers by suggesting relevant identifiers and APIs during coding. While completions are typically derived from static analysis, their usefulness depends heavily on how they are ranked, as correct predictions buried deep in the list are rarely seen by users. Most current systems rely on handcrafted heuristics or lightweight machine learning models trained on user logs, which can be further improved to capture context information and generalize across projects and coding styles. In this work, we propose a new scoring approach to ranking static completions using language models in a lightweight and model-agnostic way. Our method organizes all valid completions into a prefix tree and performs a single greedy decoding pass to collect token-level scores across the tree. This enables a precise token-aware ranking without needing beam search, prompt engineering, or model adaptations. The approach is fast, architecture-agnostic, and compatible with already deployed models for code completion. These findings highlight a practical and effective pathway for integrating language models into already existing tools within IDEs, and ultimately providing smarter and more responsive developer assistance.**

***Index Terms*—Ranking, Code Suggestions, Language Models, Integrated Development Environments, IDEs**

I. Introduction

Code completion features are among the most frequently used functionalities in modern Integrated Development Environments (IDEs) [1]–[3]. Despite the growing capabilities of Large Language Models (LLMs) to generate large code snippets [4], developers continue to rely heavily on token-level completion [5], [6] due to its speed, precision, and seamless integration within the coding workflow. It assists developers by suggesting relevant identifiers, functions, or API elements during typing, helping reduce effort, minimize errors, and navigate large codebases more effectively. The precision and responsiveness of the completion systems play a critical role in improving developer productivity and code quality.

Most IDEs generate completion candidates using static analysis [7], [8], which efficiently extracts valid suggestions from the program state. However, the effectiveness of these systems depends not only on which completions are retrieved but also on how they are ranked. For example, JetBrains IDEs rely on a proprietary machine learning model trained on anonymized usage logs to sort static completions [9]. Although these models are lightweight and efficient, they rely on handcrafted features and are largely blind to the broader semantic context of the source file. As a result, ranking decisions are often based on metadata such as selection history, symbol frequency, or usage patterns, *rather than the actual semantics of the surrounding code.*

LLMs offer a new opportunity to enhance this ranking step. While primarily used for generative tasks, LLMs also expose rich token-level probability distributions that can be repurposed to score and rank static completion candidates. However, their application in this context remains underexplored, particularly under the runtime constraints typical of local development tools.

We propose TreeRanker, a lightweight decoding-time sorting mechanism that uses an LLM to rank a predefined set of static completions. These completions are organized into a *completion tree*, where each path corresponds to a valid token sequence that represents an identifier. As the model performs greedy decoding, we collect token-level probabilities not only for the decoded path but also for all valid alternatives reachable in the tree. This allows us to construct a fine-grained ranking signal without requiring beam search [10], [11] or prompt augmentation [12], [13].

TreeRanker is fast, model-agnostic, and non-intrusive. It does not require modifying the model or retraining and operates using a single greedy decoding loop. A lightweight masking strategy is used to restrict generation to valid continuations, although we find that the primary performance gains arise from the structured score collection itself, not from hard constraints. TreeRanker is designed to work in perfect synergy with code generation models already in use within IDEs [14].

Unlike approaches that improve code completion through prompt augmentation [15], retrieval-augmented generation [16], or multi-step ranking [17], our method operates entirely within the model's native decoding process. It avoids the need for large prompts or multiple passes, making it highly compatible with already implemented single-pass inference pipelines. This design enables effective ranking even with compact language models. We demonstrate that our approach performs competitively using models as small as 135 million parameters. This makes our solution viable for integration into local development environments without sacrificing latency or ranking quality. We evaluated our method on two benchmarks that span both language diversity and the scope of completion. The first, *DotPrompts* [18], focuses on Java dereference completions and includes a mix of global APIs and locally defined

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00290

elements, providing broad coverage of typical IDE completion scenarios. The second, *StartingPoints*, is a curated subset of the Long Code Arena benchmark [19], specifically designed to target completions involving identifiers defined within the local scope of large Python projects. This refined benchmark emphasizes realistic project-specific completions and allows us to evaluate the method's ability to resolve and rank repository-specific APIs.

In summary, we investigate the following research questions:

❶ Can token-level scoring from constrained greedy decoding improve the ranking of static completions compared to existing in-IDE solutions and LLM-based baselines? ❷ Is the method efficient enough to be used in low-latency code completion scenarios?

Our contributions are:

- We propose **TreeRanker**, a novel LLM-based ranking method for static code completions that uses a decoding-time tree traversal to collect token-level scores.
- We introduce **StartingPoints**, a new dataset for evaluating completion ranking on locally defined identifiers, based on the Long Code Arena benchmark.
- We evaluate the method in global and project-specific completion scenarios using Java and Python datasets to show robust performance across languages and scopes.
- We show that our method outperforms existing ML and heuristic ranking systems while preserving the low latency of greedy decoding.

II. Background

Autoregressive Decoding. LLMs generate text and code using *autoregressive decoding* [20], producing one token at a time based on the previously generated sequence.

At each step, given a context $x = (x_1, \ldots, x_t)$, the model predicts a distribution over the vocabulary $\mathcal{V}$ for the next token:

$$P(x_{t+1} \mid x_1, \ldots, x_t) \tag{1}$$

The decoding strategy defines how tokens are selected from this distribution. In *greedy decoding*, the model chooses at each step the token with the highest probability, offering speed and determinism, but potentially missing globally optimal sequences. *Beam search*, by contrast, explores multiple candidate sequences in parallel and retains k beams by cumulative log-probability, improving exploration for a better quality result at the cost of increased latency and computational overhead [10], [11].

Constrained Decoding. To improve both the efficiency and correctness of the decoding process, we employ a grammar-constrained decoding strategy based on token masking [21]. At each step, we restrict the model's output space to only those tokens that are valid continuations from the current token position. This is implemented via a binary mask $m \in \{0,1\}^{|\mathcal{V}|}$ over the vocabulary $\mathcal{V}$, applied to the model's logits. The modified logits $\tilde{L}$ are defined as:

$$\tilde{L}_t = \begin{cases} L_t & \text{if } m_t = 1 \\ -\infty & \text{otherwise} \end{cases} \tag{2}$$

This ensures that the model cannot assign probability mass to invalid continuations. In addition, we define the mask to allow only completion-ending tokens (i.e., tokens that can terminate an identifier) when the current token position corresponds to a terminal point for one or more completions.

Static Analysis. To extract valid completions for identifier prediction, we rely on static program analysis, a well-established technique for reasoning about source code without execution [7], [8]. In our setting, we focus on predicting the next identifier following a dereference operator, such as `.` in Java or Python. IDEs and language servers support incremental parsing, which allows the construction of partial Abstract Syntax Trees (ASTs) containing placeholder nodes (e.g., `[UNKNOWN]`) to represent incomplete expressions at the cursor position. Each AST node may carry information such as variable types, bindings to declarations, and visibility constraints. Once this set of possible completions is extracted, the model generation is constrained to this list of valid continuations. As a result, decoding is explicitly type-constrained: the model is only allowed to score completions that are both syntactically valid and semantically consistent with the program's type context.

III. Methodology

Our objective is to design a code completion ranking method that can enhance the quality of suggestions in real-world development environments. This imposes two key constraints: (i) the method must remain effective even when used with small language models without relying on extended context windows; (ii) the total latency of inference and ranking must fall within the strict time budgets expected for basic code completion in IDEs [14], [22]. With this in mind, we propose a new ranking methodology *TreeRanker*, Figure 1 illustrates an example of the ranking process. It leverages off-the-shelf language models in their standard generative setting, using the unmodified code context to score static completion candidates, and guides the process through a prefix tree constructed via greedy tokenization of candidates derived from static analysis.

A. Completion Tree

The backbone of our approach is the *completion tree*. This tree encodes the valid completions at the current cursor position and serves as a guide for efficient traversal and scoring as decoding progresses. Let $\mathcal{V}$ denote the model's vocabulary, and let $\mathcal{C} = \{c_1, \ldots, c_n\}$ be the set of valid identifier completions. Each identifier $c_i \in \mathcal{C}$ is a string, which we tokenize using the model's greedy tokenizer τ [23] to obtain a sequence of tokens:

$$\tau(c_i) = (t_{i,1}, \ldots, t_{i,k_i}) \in \mathcal{V}^{k_i} \tag{3}$$

We organize these token sequences into a *completion tree* $\mathcal{T}$, implemented as a prefix trie:

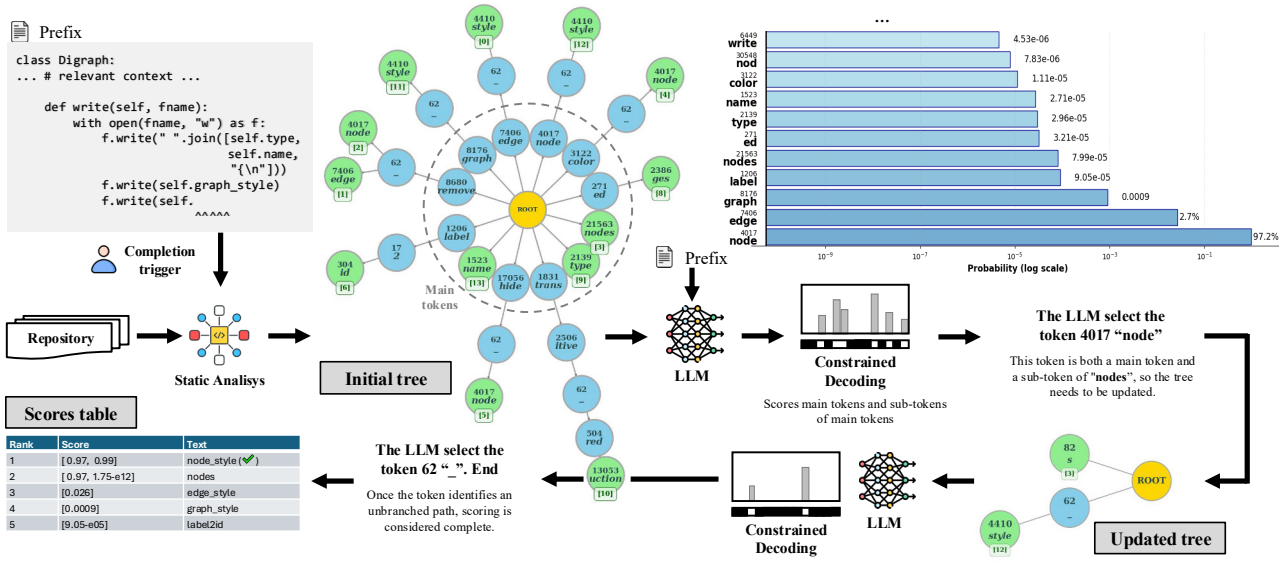

Fig. 1. Visual overview of TreeRanker scoring process over an example from StartingPoints dataset.

$$\mathcal{T} = \text{Trie}\left(\{\tau(c_i) \mid c_i \in \mathcal{C}\}\right) \tag{4}$$

Each node in the tree corresponds to a token prefix that forms a part of one or more valid completions. Nodes store a list of children, each connected by a token edge, and maintain a reference to the set of identifiers that share the current prefix. When a node represents the end of a full token sequence for an identifier, it is marked as a terminal node and stores a pointer to the corresponding completion. Because identifiers are distinct and typically do not share arbitrary suffixes, they can be cleanly organized into a prefix tree, where each path from the root to a leaf corresponds to the tokenized form of a valid completion. A key aspect of our method relies on an **optimal decoding path optimization**: *we consider that each identifier corresponds to a deterministic token sequence obtained through greedy tokenization [23], and we only fall back to more complex processing (such as handling subtokens or tokenizer artifacts) when necessary.* This allows us to minimize the number of decoding steps while still preserving the ability to recover correct completions. Our experiments show that this fallback is required in only 12% of cases with a marginal computational cost.

B. Scoring Process

During inference, let $x = (x_1, \ldots, x_m)$ denote the prefix context (i.e., the code before the cursor). At each step of tree traversal, we use the language model $\mathcal{M}$ to compute the next-token probabilities. Given the context x and a token prefix $(t_1, \ldots, t_k)$ associated with node v, we obtain:

$$\mathcal{M}(x, t_1, \ldots, t_k) = \{P(t \mid x, t_1, \ldots, t_k)\}_{t \in \mathcal{V}} \tag{5}$$

We then collect the probabilities for each valid continuation token t such that $\text{Child}(v, t) \in \mathcal{T}$. These are used to extend the token-level probability traces for each candidate identifier c_i. For every identifier $c_i \in \mathcal{C}$, we maintain a sequence of probabilities corresponding to the tokens in $\tau(c_i)$ for which we have obtained model scores:

$$\Phi(i) = (p_1, \ldots, p_{\ell_i}) \tag{6}$$

Here, ℓ_i denotes the number of tokens in $\tau(c_i)$ that have been scored so far during traversal.

One of the key advantages of the completion tree structure is the ability to detect *early stopping conditions* when the decoding path uniquely identifies a single valid completion. In many cases, only the first one or two tokens are sufficient to disambiguate the intended identifier (see Section V-D). When this occurs, decoding can be stopped early, significantly reducing the number of steps required and further minimizing the overhead introduced by the ranking system.

C. Tokenization details

As discussed earlier, our method relies on an *optimal decoding path optimization*, where each identifier is expected to follow a deterministic token sequence under greedy tokenization. This allows us to treat the completion tree as a fixed structure. However, tokens are not simple alphanumeric units but subword fragments, and distinct tokens may share common prefixes. This can introduce ambiguity in the decoding process that cannot be resolved by token identity alone. For example, if `is` and `isEmpty` are treated as distinct tokens, selecting `is` does not exclude `isEmpty` as a possible continuation. To address this, we define *main tokens* as the sequence returned by greedy tokenization for each identifier. Each *main token* represents the longest valid match at its position in the string. We then define *subtokens* as all valid tokens that that are strict prefixes of a *main token* but are not selected by

the greedy tokenizer. These relationships are tokenizer-specific and allow us to build bidirectional mappings between *main tokens* and their corresponding *subtokens*. At decoding time, we use these mappings in three key ways:

(i) We apply token-level constraints through logit masking (see Section II), enabling both *main tokens* and *subtokens* to be treated as valid candidates at each decoding step. This improves the decoding fluency, preserving as much as possible the model distribution during generation. For example, if only `isEmpty` is a valid completion, we still allow the selection of `is` without operating directly with the tokenizer.

(ii) If a *subtoken* is selected during decoding (e.g., `is`), which means that it is assigned a high probability under the model's distribution as defined in Equation 5, we interpret it as a prefix of one or multiple *main tokens*. In response, if it is a shared prefix of multiple *main tokens*, we dynamically restructure the tree by introducing the *subtoken* as a new intermediate node. All affected completions are reassigned to this node, and their suffixes are re-tokenized (removing the decoded prefix) to generate the updated branches. If the predicted *subtoken* corresponds unambiguously to a single *main token*, we move the selection to the full *main token*. This shortcut accelerates the decoding process, though it introduces a small deviation from the model decoding distribution.

(iii) Finally, once the tree is updated, we continue to decode the new node and update the probability trace $\Phi(i)$ for each candidate using the probability assigned to the selected *subtoken* (as in Equation 6). This ensures that the scoring remains consistent with the actual output of the model, while maintaining compatibility with the constrained decoding and classification strategy described in Equation 7.

D. Ranking Strategy

Identifiers are ranked by the extent to which their token sequence has been explored in the tree, i.e., by ℓ_i. Among identifiers with the same ℓ_i, we use the last available token probability as a tie-breaker. Formally:

$$\text{Rank}(c_i) < \text{Rank}(c_j) \iff (\ell_i, \Phi(i)[\ell_i]) > (\ell_j, \Phi(j)[\ell_j]) \tag{7}$$

This scoring scheme favors completions whose token paths align most closely with the greedy decoding trajectory of the model. At the same time, it captures the local confidence of the model at each branching point, even when a candidate diverges from the path of highest probability. By collecting probabilities not only for the selected tokens, but also for all valid continuations at each visited node, we retain a broader view of the model preferences and increase robustness to small decoding deviations. We also experimented with alternative scoring schemes that aggregate probabilities across multiple *subtokens* associated with a given *main token*, but found that assigning the probability of the *main token* directly yielded the best empirical results. We leave a more systematic exploration of subtoken-level score aggregation as future work.

IV. Experiment Setup

A. Benchmarks

To rigorously evaluate our ranking system for code completion, we use two different benchmarks. First, we employ the **DotPrompts** [18] benchmark. Since this dataset includes both standard library APIs and user-defined elements, it is well suited to evaluate performance on global and local identifier prediction. To focus more specifically on completions involving locally defined APIs, we introduce a subset of **Long Code Arena** [19], a benchmark for project-wide code completion. This subset, which we refer to as **StartingPoints**, consists of Python files rich in user-defined classes and functions.

1) DotPrompts: provides pre-extracted completion points for Java code following dereference operations (i.e., positions after the `.` operator), with a clear next-token prediction objective. We extend this dataset by applying both IntelliJ IDEA and VSCode completion engines at each completion point to extract ranked lists of suggestions. We report descriptive statistics in Table I to characterize the overlap and diversity between IntelliJ and VSCode completions.

TABLE I
COMPARISON STATISTICS BETWEEN INTELLIJ AND VSCODE

Metric	DotPrompts
Dataset Coverage	
Total examples	7332
Avg. list length (IntelliJ)	60.5
Avg. list length (VSCode)	60.3
Median list length (IntelliJ)	37
Median list length (VSCode)	34
Overlap Between Systems	
Avg. Jaccard Similarity	0.85
Identical completions	2859
Completely different (no overlap)	0
Significant diff. (similarity ¡ 0.5)	9.75%
Engine-Specific Completions	
Avg. completions only in IntelliJ	6.7
Avg. completions only in VSCode	5.2

2) Long Code Arena: was not originally designed for next-identifier prediction. To adapt it to our setting, we introduce **StartingPoints**, a filtered and annotated subset of Long Code Arena, specifically constructed to evaluate ranked completions of locally defined identifiers. We begin by analyzing each repository using the `tree-sitter` [24] parser to locate dereference positions, that is, all occurrences of the `.` operator. For each of these locations, we use the `Jedi` [25] static analysis library, to resolve the type of the dereferenced object. To restrict the benchmark to project-local completions, we retain only those suggestions whose definitions are found within the same repository. Unlike the DotPrompts setting, we extend the dataset using only IntelliJ completions, as it was not feasible to obtain equivalent ranked suggestions from Visual Studio Code. To ensure consistent and meaningful evaluation, we apply two key filtering steps. First, we discard

examples where the ground-truth identifier is not present in the candidate list of IntelliJ, or where IntelliJ returns fewer than five valid completions, excluding boilerplate entries such as Python dunder methods (e.g., `__del__`). Second, we exclude completion points where the ground-truth identifier begins with an underscore (`_`). This decision is driven by tokenization artifacts: tokenizers can treat tokens like `._` as atomic units, merging the dot and the underscore into a single token. Since our decoding setup assumes the dot is part of the static prefix, this behavior introduces inconsistencies and biases the model toward alphanumeric completions. Our inspection of tokenizer vocabularies confirmed that such merged tokens exist only for underscore-prefixed completions (e.g., `._`, `.___`), while other common identifier prefixes (e.g., `.f`, `.init`) are not tokenized similarly. As a result, underscore-prefixed completions are harder for the model to reach and are excluded from our evaluation. These excluded completion points represent only a minimal fraction of the dataset, and we leave a more principled handling of such cases to future work.

TABLE II
KEY STATISTICS FOR THE DOTPROMPTS AND STARTINGPOINTS.

Metric	DotPrompts	StartingPoints
Total examples	7332	1487
Unique repositories	94	60
Unique files	776	125
Avg. prefix length (lines)	151.6	288.7
Median prefix length (lines)	96	206.0
Avg. list length (IntelliJ)	60.19	53.3
Median list length (IntelliJ)	37	39

B. Metrics

Quality Metrics. To assess the quality of different ranking approaches, we employ a comprehensive set of evaluation metrics. We report *Recall@K* for $K \in \{1, 5, 20\}$, which measures the proportion of times the correct completion appears within the top-K suggestions. We also include *Mean Reciprocal Rank (MRR)*, which captures the average inverse rank of the first correct suggestion.

Among the evaluated metrics, ***MRR*** *and* ***Recall@5*** *stand out as particularly relevant for the intended use case.* Since developers typically focus on the top few suggestions in an IDE, success is defined not only by retrieving the correct identifier but also by ranking it highly in the suggestion list.

Performance Metrics. We report two metrics to assess the latency of our approach: the total inference time and the *ranking time*, defined as the interval between the generation of the first token and the end of the decoding process. The ranking time isolates the runtime impact of our ranking strategy, excluding the time required to produce the first token, which is highly sensitive to various factors, including hardware-specific optimizations, implementation details, and cache policies. Since optimizing this initial latency is outside the scope of our work and largely orthogonal to the proposed method, we adopt a conservative fixed estimate of 75 milliseconds [9] for the first token latency, consistent with previous measurements from ONNX Runtime [26] and llama.cpp [27] implementations on consumer hardware. By focusing on the time elapsed between the generation of the first token and the end of decoding, we isolate the performance impact of the ranking system and obtain a more accurate estimate of its overhead in a real IDE deployment. To quantify efficiency, we define the *Token Efficiency Ratio*, as the ratio between the number of tokens required to represent the ground truth identifier and the number of tokens generated by the model. Values higher than 1 indicate more optimal decoding, while values below 1 suggest over-generation. This metric captures the alignment between the model's decoding path and the target identifier, and reflects the efficiency of the constrained traversal.

C. Model Selection

As previously discussed, the goal of this work is to develop methods suitable for scale deployment on consumer hardware. For this reason, we focus on small and extremely compact open-source models, aiming to approximate the performance of custom deployments feasible on end-user hardware. In particular, we take inspiration from IntelliJ's production setup [14], which imposes strict constraints on memory footprint and inference time. Their completion engine leverages a quantized LLaMA-like [28] model with approximately 100M parameters and a maximum context window of $1,536$ tokens.

Motivated by these practical considerations, our study focuses on models ranging from 130M to 1.3B parameters. The selected models are: `SmolLM2-135M` [29], `SmolLM2-360M` [29] `codegen-350M-multi` [30], and `deepseek-coder-1.3b-base` [4]. We also include the SmolLM models in our study, despite their general-purpose nature, because they represent a unique class of extremely compact models. Although not exclusively trained for code, they are among the few recent models of this scale that have been exposed to coding tasks which makes them a valuable point of comparison.

To ensure a fair comparison across different model sizes, we standardize the context window to 1920 tokens for all experiments. This decision is crucial given that the ranking performance is highly sensitive to the amount of contextual information provided. We select models with consistent tokenization of the dot operator (see Section IV-A2); however, our methodology is general and can be adapted to any tokenization scheme by modifying the subset of target tokens.

D. Baselines

To provide concrete points of comparison for our ranking task, we include as baselines the code completion systems of the two most widely used IDEs: IntelliJ IDEA [31] for both the datasets and Visual Studio Code [32] for DotPrompts (Java).

IntelliJ IDEA as described by Bibaev et al. [9], uses a machine learning-based ranking model trained on anonymized usage logs collected from real developers. The model is implemented using *CatBoost* [33] with the *QuerySoftMax* [34]

loss function, which is specifically designed to prioritize the correct suggestions at the top of the completion list.

VSCode leverages IntelliCode [35], a plugin that augments the standard completion engine by reordering suggestions using machine learning. While the internal architecture of IntelliCode is not publicly documented in detail, it is known to personalize suggestions based on user behavior and code context, blending statistical insights with conventional ranking strategies.

In addition to IDE-based systems, we incorporate LLMs as baselines for code completion ranking, drawing inspiration from prior work (see Section VII) that leverages beam search outputs to guide code suggestions [36], [37]. Specifically, we generate candidate completions using greedy decoding and beam search with widths of 5 and 20. We included a filtered variant of each beam setting, denoted as *Beam@K Filtered*, where only completions present in the IntelliJ candidate list are retained. Although similar filtering could be applied using VSCode completions, our empirical analysis revealed that IntelliJ and VSCode yield comparable candidate sets. To represent the upper limit of beam-based approaches, we introduce a powerful baseline denoted as *Beam@All*. It performs an exhaustive, constrained search of the completion tree, scoring each candidate by its cumulative log-probability with a length penalty. This process makes *Beam@All efficient and functionally equivalent to a re-ranker driven directly by the model loss function*.

E. Hardware Setup

All experiments were performed on a machine equipped with a single NVIDIA RTX 4090 GPU with 24GB of memory. Inference was executed using standard PyTorch and Hugging Face Transformers [38] and in half-precision (FP16). Given this setup, our reported inference times should be interpreted as conservative upper bounds. In practice, significant reductions in latency could be achieved through quantization and hardware-aware optimizations, particularly for deployment in real-time IDE environments.

V. Results

A. Evaluation

To evaluate the effectiveness of our approach, we compare TreeRanker to standard IDE completion engines and LLM-based decoding strategies across a range of models. The results are reported in Table III. Across all model sizes and both datasets, TreeRanker consistently delivers top-tier ranking performance. Outperforms IntelliJ, VSCode, and base beam decoding variants in all retrieval metrics. For instance, on *DotPrompts*, TreeRanker on SmolLM2-135M improves the mean reciprocal rank (MRR) by up to 16 points and Recall@5 by 8 points compared to IntelliJ.

Most notably, TreeRanker matches the performance of the strongest baseline, *Beam@All*, which computes a full beam score for each candidate. The two methods yield nearly identical results (within a few percentage points) across MRR, R@5, and R@20 on both datasets and across all model scales.

TABLE III
Evaluation metrics across baseline IDEs and LLM-based completion strategies on **DotPrompts** and **StartingPoints**.

		StartingPoints				**DotPrompts**			
Model	**Method**	**MRR**	**R@1**	**R@5**	**R@20**	**MRR**	**R@1**	**R@5**	**R@20**
IDEs	VSCode	-	-	-	-	0.45	0.32	0.58	0.86
	IntelliJ	0.49	0.33	0.68	0.90	0.60	0.46	0.79	0.92
135M	Greedy	0.56	0.56			0.62	0.62		
	Beam@5	0.61	0.57	0.67		0.67	0.63	0.75	
	+ Filter	0.62	0.58	0.67		0.70	0.66	0.75	
	Beam@20	0.62	0.57	0.68	0.72	0.68	0.62	0.76	0.81
	+ Filter	0.65	0.59	0.72	0.72	0.73	0.68	0.80	0.81
	Beam@All	0.72	0.60	0.87	0.98	0.76	0.66	0.89	0.98
	TreeRanker	0.73	0.62	0.88	0.97	0.76	0.67	0.87	0.97
350M	Greedy	0.64	0.64			0.72	0.72		
	Beam@5	0.69	0.65	0.75		0.77	0.72	0.83	
	+ Filter	0.71	0.67	0.75		0.79	0.76	0.83	
	Beam@20	0.69	0.65	0.75	0.78	0.78	0.72	0.84	0.88
	+ Filter	0.72	0.68	0.78	0.78	0.82	0.77	0.87	0.88
	Beam@All	0.76	0.67	0.88	0.97	0.84	0.76	0.94	0.99
	TreeRanker	0.78	0.70	0.89	0.95	0.83	0.77	0.91	0.97
360M	Greedy	0.63	0.63			0.69	0.69		
	Beam@5	0.67	0.64	0.72		0.75	0.70	0.81	
	+ Filter	0.69	0.65	0.72		0.77	0.73	0.81	
	Beam@20	0.68	0.64	0.73	0.77	0.75	0.70	0.82	0.86
	+ Filter	0.71	0.66	0.76	0.77	0.79	0.75	0.85	0.86
	Beam@All	0.77	0.68	0.89	0.98	0.83	0.75	0.94	0.99
	TreeRanker	0.78	0.69	0.90	0.98	0.82	0.75	0.90	0.97
1.3B	Greedy	0.71	0.71			0.78	0.78		
	Beam@5	0.75	0.72	0.79		0.82	0.78	0.88	
	+ Filter	0.76	0.74	0.79		0.85	0.82	0.88	
	Beam@20	0.75	0.72	0.79	0.82	0.83	0.78	0.88	0.91
	+ Filter	0.78	0.75	0.82	0.82	0.86	0.83	0.91	0.91
	Beam@All	0.83	0.75	0.93	0.99	0.88	0.80	0.96	0.99
	TreeRanker	0.84	0.78	0.92	0.99	0.88	0.83	0.94	0.99

These results demonstrate that TreeRanker achieves ranking quality equivalent to a full scoring pass over all candidates.

A breakdown by model scale reveals an expected trend: larger models (e.g., DeepSeek-Coder 1.3B) obtain the best absolute performance. However, smaller models such as SmolLM2-135M still benefit meaningfully from TreeRanker and achieve competitive results in both datasets. The *StartingPoints* dataset presents an additional challenge due to its exclusive focus on identifiers defined within the project scope. Despite the lack of visibility on local implementations, TreeRanker maintains consistent improvements over baselines.

Beam@5-20 offers reasonable performance in Recall@1, particularly with larger models. However, when evaluating ranking quality with more informative metrics such as MRR and Recall@5, TreeRanker clearly outperforms such metrics. Moreover, increasing the beam width from 5 to 20 yields only marginal gains, and some models exhibit performance saturation. In contrast, TreeRanker's improvements reflect its ability to score and prioritize completions more accurately on the entire suggestion list.

B. Evaluation on Unseen Identifiers

We further investigate model performance in a more challenging setting and focus on completions where the correct

TABLE IV
PERFORMANCE ON IDENTIFIERS NOT PRESENT IN THE FILE PREFIX

		StartingPoints				DotPrompts			
Model	**Method**	**MRR**	**R@1**	**R@5**	**R@20**	**MRR**	**R@1**	**R@5**	**R@20**
IDEs	IntelliJ	0.26	0.10	0.40	0.92	0.50	0.37	0.66	0.89
135M	Beam@20+F	0.20	0.18	0.23	0.23	0.49	0.42	0.59	0.60
	Beam@All	0.45	0.28	0.66	0.95	0.59	0.43	0.79	0.95
	TreeRanker	0.47	0.30	0.71	0.94	0.58	0.44	0.76	0.94
350M	Beam@20+F	0.28	0.24	0.32	0.32	0.64	0.57	0.73	0.74
	Beam@All	0.47	0.30	0.68	0.92	0.72	0.59	0.87	0.97
	TreeRanker	0.49	0.33	0.69	0.88	0.69	0.58	0.83	0.96
360M	Beam@20+F	0.25	0.21	0.29	0.29	0.60	0.53	0.68	0.69
	Beam@All	0.48	0.32	0.68	0.94	0.70	0.57	0.87	0.97
	TreeRanker	0.49	0.33	0.72	0.96	0.67	0.55	0.82	0.95
1.3B	Beam@20+F	0.36	0.32	0.39	0.40	0.73	0.67	0.81	0.81
	Beam@All	0.58	0.43	0.79	0.98	0.78	0.67	0.92	0.98
	TreeRanker	0.59	0.45	0.78	0.98	0.77	0.68	0.89	0.98

identifier is not present in the input prefix. This subset of $2,676$ samples for *DotPromts* and 340 samples for *StartingPoints* captures a common scenario in which the model must choose among static completions that are unseen in the context window, such as when accessing a class member or method for the first time. The results for this setting are reported in Table IV. As expected, all LLM-based methods exhibit a notable drop in Recall@1 compared to the full benchmark. This confirms that models rely heavily on the provided prefix to confidently resolve completions. This effect is consistent across model sizes and decoding strategies, including TreeRanker.

Even with constrained decoding, Recall@1 remains lower than in the full setting, *supporting the intuition that filtering invalid completions is necessary but not sufficient for accurate top-rank selection.* However, despite this drop in rank one, TreeRanker continues to maintain high MRR and Recall@5 across all configurations. This indicates that the scoring strategy still effectively retrieves the correct completions from the model, even when full identifier resolution is not achievable from the context alone. These gains are particularly relevant in practical scenarios, where developers often select completions from among the top few entries during first-time exploration. The trend holds across both benchmarks.

Answer RQ ❶: It consistently matches or outperforms traditional IDE systems and standard LLM baselines across all model sizes and benchmarks. TreeRanker achieves ranking quality equivalent to a full scoring pass over all candidates. Moreover, it maintains strong MRR and Recall@5 even in the challenging case where the target identifier is not present in the prefix.

C. Performance Evaluation

To assess the practicality of our approach in real-world settings, we measure the average inference time of each model and decoding strategy, separating the total decoding time from the time specifically attributable to the ranking mechanism. The results are reported in Table V for both benchmarks.

To robustly assess generation latency across configurations, we conducted five independent runs for each experimental setting, collecting per-sample inference data. For each input, we recorded both the total generation time and the **ranking time**, computed as the duration from the generation of the first token to the completion of the decoding process. As discussed previously, this excludes the initial token latency, for which we adopt a conservative upper bound estimate of 75 milliseconds based on previous interactive completion benchmarks [9].

We report the mean generation and ranking times across all runs, along with associated confidence intervals (CIs) to quantify variability and measurement precision. CIs were computed using Student's t-distribution [39]. Given the large number of completion points, $7,332$ for *DotPrompts* and $1,487$ for *LCA-StartingPoints*, we compute per-sample statistics by first averaging across the five runs at each fixed input position, then calculating the standard error and confidence interval for that position. The final reported values are obtained by averaging these per-sample means and confidence intervals across the

TABLE V
AVERAGE GENERATION AND RANKING OVERHEAD TIMES (SECONDS).

Model	**Method**	**Total Time**	**Ranking Time**	**Ranking Speedup**
StartingPoints				
SmolLM2-135M	Greedy	0.421 ± 0.005	0.101 ± 0.001	1.53x
	Beam@5	0.486 ± 0.008	0.161 ± 0.002	2.44x
	Beam@20	0.743 ± 0.008	0.418 ± 0.002	6.34x
	Beam@All	1.652 ± 0.083	1.615 ± 0.082	24.47x
	TreeRanker	**0.392 ± 0.011**	**0.066 ± 0.003**	∼
CodeGen-350M	Greedy	0.602 ± 0.005	0.240 ± 0.002	1.60x
	Beam@5	0.755 ± 0.009	0.426 ± 0.004	2.84x
	Beam@20	1.915 ± 0.018	0.483 ± 0.016	3.22x
	Beam@All	1.457 ± 0.052	1.426 ± 0.051	9.51x
	TreeRanker	**0.522 ± 0.011**	**0.150 ± 0.003**	∼
SmolLM2-360M	Greedy	0.494 ± 0.005	0.157 ± 0.001	1.57x
	Beam@5	0.598 ± 0.007	0.258 ± 0.002	2.58x
	Beam@20	1.201 ± 0.007	0.797 ± 0.002	7.97x
	Beam@All	1.816 ± 0.110	1.777 ± 0.108	17.77x
	TreeRanker	**0.446 ± 0.013**	**0.100 ± 0.003**	∼
DeepSeek-Coder 1.3B (w. Flash Attention)	Greedy	0.374 ± 0.003	0.139 ± 0.001	1.55x
	Beam@5	0.492 ± 0.004	0.124 ± 0.002	1.38x
	Beam@20	1.120 ± 0.008	0.220 ± 0.002	2.45x
	Beam@All	1.583 ± 0.073	1.535 ± 0.072	17.06x
	TreeRanker	**0.328 ± 0.007**	**0.090 ± 0.003**	∼
DotPrompts				
SmolLM2-135M	Greedy	0.398 ± 0.008	0.070 ± 0.002	1.27x
	Beam@5	0.443 ± 0.006	0.110 ± 0.002	2.00x
	Beam@20	0.627 ± 0.006	0.295 ± 0.002	5.36x
	Beam@All	1.682 ± 0.028	1.644 ± 0.027	29.9x
	TreeRanker	**0.330 ± 0.023**	**0.055 ± 0.005**	∼
CodeGen-350M	Greedy	0.505 ± 0.095	0.126 ± 0.018	1.30x
	Beam@5	0.642 ± 0.009	0.290 ± 0.004	2.99x
	Beam@20	1.448 ± 0.015	0.499 ± 0.011	5.14x
	Beam@All	1.604 ± 0.022	1.562 ± 0.022	16.98x
	TreeRanker	**0.392 ± 0.009**	**0.097 ± 0.002**	∼
SmolLM2-360M	Greedy	0.430 ± 0.008	0.092 ± 0.001	1.21x
	Beam@5	0.520 ± 0.014	0.181 ± 0.004	2.38x
	Beam@20	0.952 ± 0.007	0.571 ± 0.002	7.51x
	Beam@All	1.463 ± 0.030	1.432 ± 0.030	18.84x
	TreeRanker	**0.421 ± 0.008**	**0.076 ± 0.002**	∼
DeepSeek-Coder 1.3B (w. Flash Attention)	Greedy	0.318 ± 0.014	0.080 ± 0.004	1.31x
	Beam@5	0.400 ± 0.005	0.076 ± 0.001	1.25x
	Beam@20	0.850 ± 0.005	0.131 ± 0.001	2.15x
	Beam@All	1.835 ± 0.024	1.794 ± 0.023	29.41x
	TreeRanker	**0.257 ± 0.005**	**0.061 ± 0.002**	∼

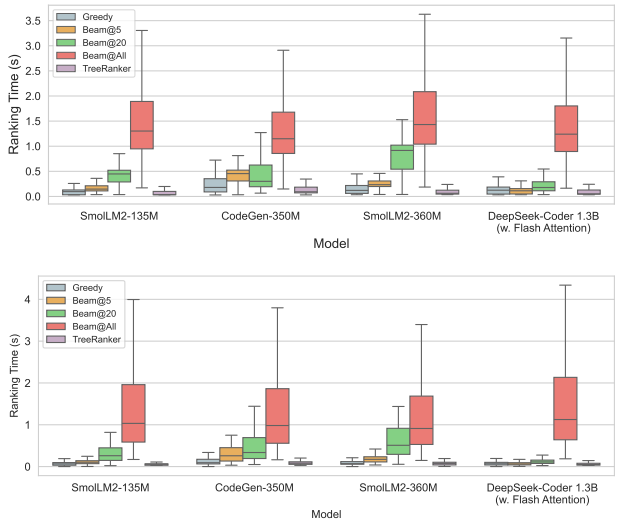

Fig. 2. Distribution of ranking times across models and decoding strategies on StartingPoints(1st) and Dotprompt(2nd). TreeRanker achieves stable, low-latency performance while maintaining high ranking perfromances.

dataset. Compared to computing a global confidence interval over per-run averages, this approach better reflects the stability of runtime behavior. The performance evaluation indicates that TreeRanker remains well within interactive latency bounds. In DotPrompts, for example, SmolLM2-135M completes the ranking in 55 ± 5 milliseconds. Adding the conservative 75ms upper bound for first token generation time, the total latency remains safely below the commonly accepted threshold for real-time task [14], [22]. On average, this performance is even faster than the model's native greedy decoding, thanks to the possibility of early stopping during generation. TreeRanker thus achieves a favorable trade-off, delivering improvements in ranking quality (as shown in Section V-A) while maintaining response times compatible with deployment in interactive environments.

*TreeRanker achieves ranking quality on par with computationally expensive reranking approaches while delivering up to **30× speedup in inference time**, combining high accuracy with substantial efficiency gains.*

To further illustrate the distribution of the ranking times between individual inputs, Figure 2 presents a boxplot for each model and decoding method. Although Table V summarizes overall performance using average values and confidence intervals across the dataset, this figure shows how the ranking times vary between different completion points. TreeRanker consistently exhibits low median latency and narrow interquartile ranges, comparable to greedy decoding, and substantially more stable than beam-based approaches. Moreover, the figure showcases the core limitation of reranking techniques with linear complexity in the number of candidates: they introduce excessive variance in latency, which becomes a major issue in human-interactive scenarios where consistency is crucial.

D. Statistics on Tree Manipulation

To better understand the internal dynamics of our decoding strategy, we analyze structural statistics related to tree traversal using the DeepSeek-Coder 1.3B model. Similar patterns were observed across all models. As shown in Table VI, most completions terminate early, with 77% in *DotPrompts* and 89% in *StartingPoints* completing before fully generating the identifier. This early exit behavior indicates that the model is often able to narrow the candidate set quickly, reducing the number of decoding steps and saving computational resources.

TABLE VI
STATISTICS ON INFERENCE WITH TREERANKER

Metric	DotPrompts	StartingPoints
Dataset stats		
Total examples	7332	1487
Avg. ground truth length (tokens)	3.07	4.47
Median ground truth length (tokens)	3	4
TreeRanker Features		
Early completion	77%	89%
Gen. new tree sub-branch	13%	9%
Main Tokens push	1.4%	1.3%
Generation		
Avg. gen. tokens	**1.86**	**2.15**
Std. gen. tokens	0.84	1.77
Single forward pass	35%	57%
Within two forward passes	84%	68%
Token Efficiency Ratio ↑	1.85	2.77

Cases where the tree must be modified to resolve ambiguities between overlapping token prefixes are rare. These cases account for just 13% of completions in *DotPrompts* and 9% in *StartingPoints*, confirming that most decoding paths align well with the optimistic tree structure built from greedy tokenization. Additionally, main token pushes are observed in only about 1.3% to 1.4% of completions. These low rates reflect the stability and structural alignment of the decoding process. Overall, the method proves to be highly efficient in practice, with average Token Efficiency Ratios of 1.85 and 2.77 on the two datasets, showing that most completions are ranked correctly with significantly fewer decoding steps than the full tokenized length of the target identifier.

Answer RQ ❷: TreeRanker is capable of ranking full candidate lists with high precision by extracting full knowledge of the model using only 1.86 and 2.15 tokens on average per completion. It matches the ranking quality of computationally expensive reranking approaches while achieving up to a 30× speedup in inference time. These properties make TreeRanker well suited for interactive code completion scenarios.

E. Ablation Study

Constrained decoding is not introduced as a core contribution, but rather as an optimization to reduce the number of decoding steps during prefix tree traversal. Crucially, it is not required for the ranking itself, which is solely based on the token probabilities of the model. As shown in Table VII,

TABLE VII
ABLATION STUDY: IMPACT OF CONSTRAINED DECODING ON RANKING PERFORMANCE

		StartingPoints				DotPrompts			
Model	**Const.**	**EM**	**MRR**	**R@1**	**R@5**	**EM**	**MRR**	**R@1**	**R@5**
SmolLM2-135M	✓	0.62	0.73	0.62	0.88	0.67	0.76	0.67	0.87
	✗	0.56	0.72	0.61	0.87	0.62	0.75	0.66	0.87
CodeGen-350M	✓	0.70	0.78	0.70	0.89	0.77	0.83	0.77	0.91
	✗	0.64	0.77	0.69	0.89	0.72	0.83	0.76	0.91
SmolLM2-360M	✓	0.69	0.78	0.69	0.90	0.75	0.82	0.75	0.90
	✗	0.63	0.78	0.68	0.90	0.69	0.81	0.74	0.90
DeepSeek-Coder 1.3B	✓	0.78	0.84	0.78	0.92	0.83	0.88	0.83	0.94
	✗	0.71	0.84	0.77	0.92	0.78	0.87	0.82	0.94

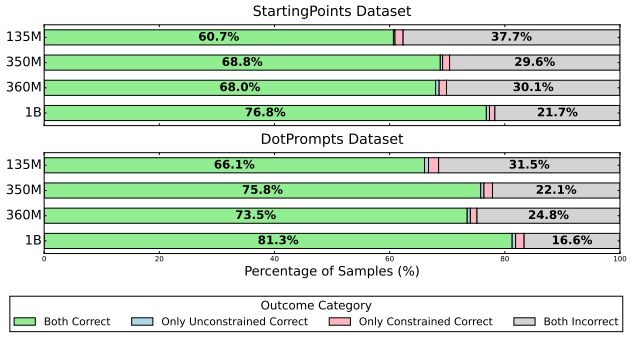

Fig. 3. Recall@1 comparison of model outputs w/ and w/o constrained dec.

performance in MRR and recall metrics remains effectively unchanged with / and without constraints. Figure 3 further shows that both variants recover largely overlapping sets of correct completions, confirming that the quality of the ranking stems primarily from the novel scoring mechanism itself. The new metric Exact Match (EM) is representative of the actual generated output of the LLM. This metric matches Recall@1 when the generation is controlled.

This important result demonstrates that TreeRanker can be applied effectively even in scenarios where model generation cannot be controlled or constrained.

VI. DISCUSSION

Our results (Section V-A) demonstrate that TreeRanker consistently outperforms both traditional IDE ranking systems and common LLM baselines, matching the performance of computationally expensive reranking techniques while achieving up to a 30× speedup, across all tested model sizes and datasets. In *DotPrompts*, our method achieves up to +16 MRR and +8 Recall@5 points over the best commercial IDE performance evem with the smallest model. Similar gains are observed in *StartingPoints*, where completions target project-local identifiers, a setting where many neural systems degrade sharply due to lack of context.

Even under challenging conditions (See Table IV), where the correct identifier is unseen in the prefix, TreeRanker maintains high MRR and Recall@5. This indicates that our approach can still effectively distinguish semantically plausible candidates. This contrasts sharply with beam search baselines, which flatten in performance as beam width increases and offer only marginal ranking benefits at significant computational cost. The efficiency metrics reinforce the practicality of our approach. TreeRanker reduce minimal latency compared to greedy decoding, often outperforming beam-based strategies in both runtime and accuracy. With most completions resolved in one or two forward passes, and early termination occurring in over 75% of cases, our method remains compatible with real-time usage in development environments.

In short, our method delivers substantial real-world value: better completions, faster inference, and smooth integration into existing IDE workflows. This sets a new baseline for what small models can achieve in interactive coding tools.

A. Differences between Java and Python Datasets

The structure of identifiers differs substantially between Java and Python, shaping how completions unfold during decoding. Java identifiers (*DotPrompts*) commonly follow *camelCase*, producing compact tokenizations with fewer subword units. In contrast, Python identifiers (*StartingPoints*) adopt *snake_case*, resulting in longer token sequences. This is reflected in the average ground-truth token length: 3.07 for Java versus 4.47 for Python (See VI). These differences impact decoding behavior. In *DotPrompts*, only 35% of completions are resolved in a single forward pass, yet 84% are completed within two, indicating that most Java identifiers are fully predicted with minimal decoding overhead. In *StartingPoints*, 57% complete in one step, but gains taper off with additional passes (68% within two). This suggests greater early predictability but more dispersed token structures. These patterns confirm that identifier conventions affect decoding efficiency. Despite longer ground truth lengths, the token efficiency ratio remains high across both datasets, showing that *TreeRanker* accurately prioritizes relevant completions with fewer steps than naive greedy generation.

B. Limitations and Future Directions

We designed our approach with latency-aware constraints, although we do not claim real-time performance guarantees under a fully optimized production environment. All experiments were conducted using standard transformer implementations via HuggingFace [38], without quantization or low-level runtime optimizations. As such, we do not report full end-to-end latency figures typical of production-grade systems. Nonetheless, our results show that the method achieves low-latency scoring even in this unoptimized setting. This suggests a strong potential for further improvements through integration with optimized inference runtimes and quantized models. We leave the implementation and evaluation of such a deployment to future work.

Another limitation of our current ranking strategy is the lack of explicit mechanisms to encourage diversity among top-ranked completions. In cases where multiple valid suggestions share the same prefix or semantic root, and receive identical scores from our decoding-based method, the resulting

ranked list may lack variation. One possible solution is to filter out low-confidence tokens during traversal, allowing the model to focus only on the most probable continuations and reduce noise in score accumulation. Additionally, lightweight heuristics could be introduced to promote diversity, such as limiting the number of completions that share the same initial token. In ambiguous situations where several candidates remain indistinguishable by score, the system could fall back on existing ML-based ranking components already deployed in IDEs to refine or reorder the final top-k list.

A promising direction for future work is to evaluate the performance of TreeRanker within real-world IDEs to assess its impact on developer productivity and interaction patterns in practical coding scenarios. This evaluation was beyond the scope of the current work, which focuses on model design and controlled benchmarking.

VII. Related Work

LLM-Based Completion Ranking. Proposed LLM-based ranking systems commonly rely on *beam search* to generate multiple candidate completions, followed by a separate learned model to re-rank them. The ML-enhanced code completion system from Google Research [36] and the method by Li et al. [37] follow this architecture, treating the LLM purely as a generator. These systems ignore the structure of the code and perform ranking only after generation, resulting in unnecessary decoding overhead and limited semantic precision, making those approach sub-optimal in a limited resource environment. FIRST [15] re-frames completion ranking as a prompt-based classification task, injecting the full list of completions into the prompt and requiring the LLM to select the correct one. Although this strategy shows promise in controlled settings, it *requires general-purpose NLP reasoning and fine-tuning* on the task, making it incompatible with standard code-specific models that are typically deployed. Moreover, supporting this strategy would require shipping and maintaining a second model dedicated to ranking, adding memory and compute overhead that is impractical in constrained environments. In contrast, **our method is designed to integrate directly with existing completion engines**. TreeRanker utilizes the same context as the generation model, allowing computational reuse, and is expressly model-agnostic, requiring no additional fine-tuning or separate models.

ML-based Ranking. In this work, we focused on current implementations of ML-based ranking models used in the most popular IDEs on the market, using them as baselines to estimate our approach's impact. However, other equally valid methodologies have also been explored. For instance, Svyatkovskiy et al. [40] explored the use of LSTM-based models for capturing the semantic information embedded in code structure. Similarly, Asaduzzaman et al. [41] proposed a context-sensitive technique that ranks suggestions based on similarity to previous usage examples. In contrast, our approach relies on attention mechanisms to extract relevant context information, which enables the model to dynamically focus on the most informative elements for each prediction.

Monitor-Guided Decoding (MGD) [18] proposes a method for injecting static analysis constraints into the LLM decoding process. BGD is a method for controlled code generation; this fundamental difference in objectives makes a direct quantitative comparison infeasible. Their implementation relies heavily on repeated conversions between strings and token IDs to maintain consistency with the model-generated tokens and completion strings. This introduces significant complexity and runtime overhead, making the method less suitable for latency-sensitive tasks like code completion ranking.

Tree-Based Representations. Kim et al. [42] propose a transformer-based approach for code completion that explicitly incorporates syntactic structure by feeding ASTs into the model. While their method improves precision for structured code, such as known API calls, it remains limited in handling out-of-vocabulary identifiers, as the AST vocabulary is predefined and static. In contrast, our approach operates at the token level and remains compatible with open vocabularies, enabling completion of both seen and unseen identifiers.

VIII. Conclusions

Recent findings show that 81% of developers still rely on token-level completion, far exceeding 32% who use statement-level suggestions [5]. This underscores a critical point: code generation is only one aspect of modern programming assistance, while traditional token-level suggestions remain central to developer productivity. We introduced TreeRanker, a lightweight decoding-time strategy that enhances code completion by leveraging language model probabilities to more effectively rank token-level suggestions such as identifiers and APIs. Unlike conventional IDE systems that rely on handcrafted heuristics or manually engineered features [9], TreeRanker integrates seamlessly with existing language models for code generation. It requires no prompt engineering, no model retraining, and no additional inference passes. By gathering fine-grained probabilities over a structured prefix tree during a single greedy decoding step, TreeRanker produces accurate rankings at low computational cost, effectively extracting the model's full potential in just a few forward passes. Beyond its ranking accuracy, TreeRanker meets the strict latency requirements of interactive environments. It achieves response times faster than standard greedy decoding and matches the quality of expensive reranking baselines. This balance of speed and precision makes it highly suitable for deployment in real-world IDEs. Overall, our approach moves beyond isolated code generation and provides a lightweight, model-agnostic solution that bridges the gap between LLMs' capabilities and the low-latency demands of local IDEs tools.

Acknowledgments. This work was conducted as part of the AI for Software Engineering (AI4SE) collaboration between JetBrains and Delft University of Technology.

REFERENCES

[1] G. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the Eclipse IDE?" *IEEE Software*, vol. 23, no. 4, pp. 76–83, Jul. 2006. [Online]. Available: https://ieeexplore.ieee.org/document/1657944

[2] S. Amann, S. Proksch, S. Nadi, and M. Mezini, "A Study of Visual Studio Usage in Practice," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, Mar. 2016, pp. 124–134. [Online]. Available: https://ieeexplore.ieee.org/document/7476636

[3] M. Izadi, J. Katzy, T. Van Dam, M. Otten, R. M. Popescu, and A. Van Deursen, "Language models for code completion: A practical evaluation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[4] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence," Jan. 2024, arXiv:2401.14196 [cs]. [Online]. Available: http://arxiv.org/abs/2401.14196

[5] C. Wang, J. Hu, C. Gao, Y. Jin, T. Xie, H. Huang, Z. Lei, and Y. Deng, "Practitioners' expectations on code completion," *arXiv preprint arXiv:2301.03846*, 2023.

[6] M. Izadi, R. Gismondi, and G. Gousios, "Codefill: Multi-token code completion by jointly learning from structure and naming sequences," in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 401–412.

[7] H. Pei, J. Zhao, L. Lausen, S. Zha, and G. Karypis, "Better context makes better code language models: a case study on function call argument completion," in *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'23/IAAI'23/EAAI'23. AAAI Press, 2023. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1609/aaai.v37i4.25653

[8] D. Shrivastava, H. Larochelle, and D. Tarlow, "Repository-level prompt generation for large language models of code," in *Proceedings of the 40th International Conference on Machine Learning*, ser. ICML'23. JMLR.org, 2023.

[9] V. Bibaev, A. Kalina, V. Lomshakov, Y. Golubev, A. Bezzubov, N. Povarov, and T. Bryksin, "All You Need Is Logs: Improving Code Completion by Learning from Anonymous IDE Usage Logs," Sep. 2022, arXiv:2205.10692 [cs]. [Online]. Available: http://arxiv.org/abs/2205.10692

[10] M. Freitag and Y. Al-Onaizan, "Beam Search Strategies for Neural Machine Translation," in *Proceedings of the First Workshop on Neural Machine Translation*, 2017, pp. 56–60, arXiv:1702.01806 [cs]. [Online]. Available: http://arxiv.org/abs/1702.01806

[11] L. Huang, K. Zhao, and M. Ma, "When to Finish? Optimal Beam Search for Neural Text Generation (modulo beam size)," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, M. Palmer, R. Hwa, and S. Riedel, Eds. Copenhagen, Denmark: Association for Computational Linguistics, Sep. 2017, pp. 2134–2139. [Online]. Available: https://aclanthology.org/D17-1227/

[12] W. Fan, Y. Ding, L. Ning, S. Wang, H. Li, D. Yin, T.-S. Chua, and Q. Li, "A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models," in *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. Barcelona Spain: ACM, Aug. 2024, pp. 6491–6501. [Online]. Available: https://dl.acm.org/doi/10.1145/3637528.3671470

[13] T. Ahmed, K. S. Pai, P. Devanbu, and E. Barr, "Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization)," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 1–13. [Online]. Available: https://dl.acm.org/doi/10.1145/3597503.3639183

[14] A. Semenkin, V. Bibaev, Y. Sokolov, K. Krylov, A. Kalina, A. Khannanova, D. Savenkov, D. Rovdo, I. Davidenko, K. Karnaukhov, M. Vakhrushev, M. Kostyukov, M. Podvitskii, P. Surkov, Y. Golubev, N. Povarov, and T. Bryksin, "Full Line Code Completion: Bringing AI to Desktop," Oct. 2024, arXiv:2405.08704 [cs]. [Online]. Available: http://arxiv.org/abs/2405.08704

[15] R. Gangi Reddy, J. Doo, Y. Xu, M. A. Sultan, D. Swain, A. Sil, and H. Ji, "FIRST: Faster improved listwise reranking with single token decoding," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, Eds. Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024, pp. 8642–8652. [Online]. Available: https://aclanthology.org/2024.emnlp-main.491/

[16] Y. Chen, C. Gao, M. Zhu, Q. Liao, Y. Wang, and G. Xu, " APIGen: Generative API Method Recommendation ," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Los Alamitos, CA, USA: IEEE Computer Society, Mar. 2024, pp. 171–182. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SANER60148.2024.00025

[17] Z. Qin, R. Jagerman, K. Hui, H. Zhuang, J. Wu, L. Yan, J. Shen, T. Liu, J. Liu, D. Metzler, X. Wang, and M. Bendersky, "Large language models are effective text rankers with pairwise ranking prompting," in *Findings of the Association for Computational Linguistics: NAACL 2024*, K. Duh, H. Gomez, and S. Bethard, Eds. Mexico City, Mexico: Association for Computational Linguistics, Jun. 2024, pp. 1504–1518. [Online]. Available: https://aclanthology.org/2024.findings-naacl.97/

[18] L. A. Agrawal, A. Kanade, N. Goyal, S. Lahiri, and S. Rajamani, "Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context," *Advances in Neural Information Processing Systems*, vol. 36, pp. 32 270–32 298, Dec. 2023. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/hash/662b1774ba8845fc1fa3d1fc0177ceeb-Abstract-Conference.html

[19] E. Bogomolov, A. Eliseeva, T. Galimzyanov, E. Glukhov, A. Shapkin, M. Tigina, Y. Golubev, A. Kovrigin, A. van Deursen, M. Izadi, and T. Bryksin, "Long Code Arena: a Set of Benchmarks for Long-Context Code Models," Jun. 2024, arXiv:2406.11612 [cs]. [Online]. Available: http://arxiv.org/abs/2406.11612

[20] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is All you Need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[21] S. Geng, M. Josifoski, M. Peyrard, and R. West, "Grammar-constrained decoding for structured NLP tasks without finetuning," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 10 932–10 952. [Online]. Available: https://aclanthology.org/2023.emnlp-main.674/

[22] R. J. Kosinski, "A literature review on reaction time," *Clemson University*, vol. 10, no. 1, pp. 337–344, 2008.

[23] O. Uzan, C. W. Schmidt, C. Tanner, and Y. Pinter, "Greed is All You Need: An Evaluation of Tokenizer Inference Methods," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 813–822. [Online]. Available: https://aclanthology.org/2024.acl-short.73/

[24] "tree-sitter/tree-sitter," May 2025, original-date: 2013-11-06T06:16:00Z. [Online]. Available: https://github.com/tree-sitter/tree-sitter

[25] "API Overview — Jedi 0.19.2 documentation." [Online]. Available: https://jedi.readthedocs.io/en/latest/docs/api.html

[26] "ONNX | Home." [Online]. Available: https://onnx.ai/

[27] "ggml-org/llama.cpp," May 2025, original-date: 2023-03-10T18:58:00Z. [Online]. Available: https://github.com/ggml-org/llama.cpp

[28] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "LLaMA: Open and Efficient Foundation Language Models," Feb. 2023, arXiv:2302.13971 [cs]. [Online]. Available: http://arxiv.org/abs/2302.13971

[29] L. B. Allal, A. Lozhkov, E. Bakouch, G. M. Blázquez, G. Penedo, L. Tunstall, A. Marafioti, H. Kydlíček, A. P. Lajarín, V. Srivastav, J. Lochner, C. Fahlgren, X.-S. Nguyen, C. Fourrier, B. Burtenshaw, H. Larcher, H. Zhao, C. Zakka, M. Morlon, C. Raffel, L. v. Werra, and T. Wolf, "SmolLM2: When Smol Goes Big – Data-Centric Training of a Small Language Model," Feb. 2025, arXiv:2502.02737 [cs]. [Online]. Available: http://arxiv.org/abs/2502.02737

[30] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "CodeGen: An Open Large Language Model for Code

with Multi-Turn Program Synthesis," Feb. 2023, arXiv:2203.13474 [cs]. [Online]. Available: http://arxiv.org/abs/2203.13474
[31] "IntelliJ IDEA – the IDE for Pro Java and Kotlin Development." [Online]. Available: https://www.jetbrains.com/idea/
[32] "Visual Studio Code - Code Editing. Redefined." [Online]. Available: https://code.visualstudio.com/
[33] Z. Cao, T. Qin, T.-Y. Liu, M.-F. Tsai, and H. Li, "Learning to rank: from pairwise approach to listwise approach," in *Proceedings of the 24th international conference on Machine learning*, ser. ICML '07. New York, NY, USA: Association for Computing Machinery, Jun. 2007, pp. 129–136. [Online]. Available: https://doi.org/10.1145/1273496.1273513
[34] "Ranking: objectives and metrics |." [Online]. Available: https://catboost.ai/docs/en/concepts/loss-functions-ranking#QuerySoftMax
[35] A. Silver, "Introducing Visual Studio IntelliCode," May 2018. [Online]. Available: https://devblogs.microsoft.com/visualstudio/introducing-visual-studio-intellicode/
[36] M. Tabachnyk, "ML-Enhanced Code Completion Improves Developer Productivity." [Online]. Available: https://research.google/blog/ml-enhanced-code-completion-improves-developer-productivity/
[37] J. Li, R. Huang, W. Li, K. Yao, and W. Tan, "Toward Less Hidden Cost of Code Completion with Acceptance and Ranking Models," Jun. 2021, arXiv:2106.13928 [cs]. [Online]. Available: http://arxiv.org/abs/2106.13928
[38] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. v. Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "HuggingFace's Transformers: State-of-the-art Natural Language Processing," Jul. 2020, arXiv:1910.03771 [cs]. [Online]. Available: http://arxiv.org/abs/1910.03771
[39] Student, "The Probable Error of a Mean," *Biometrika*, vol. 6, no. 1, pp. 1–25, 1908, publisher: [Oxford University Press, Biometrika Trust]. [Online]. Available: https://www.jstor.org/stable/2331554
[40] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan, "Pythia: Ai-assisted code completion system," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2727–2735. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1145/3292500.3330699
[41] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "Cscc: Simple, efficient, context sensitive code completion," in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '14. USA: IEEE Computer Society, 2014, p. 71–80. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1109/ICSME.2014.29
[42] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code Prediction by Feeding Trees to Transformers," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, May 2021, pp. 150–162, iSSN: 1558-1225. [Online]. Available: https://ieeexplore.ieee.org/document/9402114/