Structure-Aware Corpus Construction and User-Perception-Aligned Metrics for Large-Language-Model Code Completion

Anonymous ACL submission

Abstract

Code completion technology based on large language model has significantly improved the development efficiency of programmers. However, in practical applications, there remains a gap between current commonly used code completion evaluation metrics and users' actual perception. To address this issue, we propose two evaluation metrics for code completion tasks—LCP and ROUGE-LCP, from the perspective of probabilistic modeling. Furthermore, to tackle the lack of effective structural semantic modeling and cross-module dependency information in LLMs for repository-level code completion scenarios, we propose a data processing method based on a Structure-Preserving and Semantically-Reordered Code Graph (SPSR-Graph). Through theoretical analysis and experimental validation, we demonstrate the superiority of the proposed evaluation metrics in terms of user perception consistency, as well as the effectiveness of the data processing method in enhancing model performance.

1 Introduction

In recent years, the capabilities of large language model (LLM) in code understanding and generation have made remarkable progress, greatly driving transformations in the software development field. Code assistants such as GitHub Copilot (GitHub, 2021) and Cursor (Cursor, 2023) have enabled efficient handling of "on-the-fly completion" tasks like single-line and in-line code completions, significantly boosting development efficiency (Takerngsaksiri et al., 2024; Qi et al., 2024).

Currently, LLMs perform well in code completion task with contextual information. Despite their excellent performance on synthetic datasets, only a few models have been successfully applied to real-world products. In our practical application of XXX-Code-Copilot (a private-domain code assistant for communication), we encountered similar issues, primarily attributed to the following two factors:

1. A discrepancy between model capabilities and users' actual perception, reflecting the limitations of current evaluation metrics in capturing user experience (Liu et al., 2024a; Wu et al., 2024; Van Dam et al., 2024);

2. Existing LLMs typically rely on simple token sequence-based training methods, focusing on local associations between tokens, while failing to effectively capture the intrinsic structure and cross-file, cross-module dependencies of code (Shi et al., 2023; Liu et al., 2024b).

These issues significantly limit the practical application value of code models in complex industrial environments.

To address the first issue, some researches have attempted to screen suitable evaluation metrics by incorporating user perception data and explore the relationship between evaluation metrics and user behavior (Aye et al., 2021; Bibaev et al., 2022; van Dam et al., 2023). However, in practical scenarios, the design of evaluation metrics needs to balance LLM characteristics with user perception to better guide model training. Therefore, how to design effective evaluation metrics has become a key challenge in evaluating LLMs' code completion capabilities.

Inspired by this, we propose two evaluation metrics for "on-the-fly completion" tasks: LCP (Longest Common Prefix) and ROUGE-LCP. We first theoretically analyze the relationship between these metrics and the probability distributions of model outputs; subsequently, based on XXX-Code-Copilot's data logging system, we conduct qualitative and quantitative analyses of over ten thousand user completion behaviors within two months, validating the consistency between the proposed metrics and user adoption behavior.

Regarding the second issue, existing researches enhance models' structural awareness by construct-

ing training corpora using abstract syntax trees (ASTs) (Gong et al., 2024; Jiang et al., 2025). However, these methods are limited to local structures within single files, lacking effective perception of global cross-file dependencies. Other researches adopt Retrieval-Augmented Generation (RAG) methods to explicitly retrieve dependent code snippets for target code fragments, enhancing the model's completion capabilities (Liu et al., 2024b), but fail to fully consider context and model affinity. How to perform repository-level code structure modeling and enhance models' perception of cross-file, cross-module code structures and semantic dependencies has become an urgent problem to be solved.

To this end, we propose a repository-level code corpus processing framework. First, we design an AST-based semantic unit segmentation method, enhancing the semantic structural integrity of pretraining corpora through structural consistency checks and multi-scale granularity control. Subsequently, we construct a Structure-Preserving and Semantically-Reordered Code Graph (SPSR-Graph), explicitly modeling function-level call relationships and struct reference paths across files, thereby significantly improving the model's understanding of repository-level code. Experimental results show that our proposed method performs exceptionally well in the communication domain's "on-the-fly completion" task, significantly improving the quality and generalization ability of code completion in cross-file, cross-module scenarios.

In summary, the main contributions of this paper are as follows:

- We propose two evaluation metrics for "on-the-fly completion" tasks—LCP and ROUGE-LCP. Through theoretical derivation and empirical analysis, we demonstrate the consistency of the proposed metrics with user perception.

- We propose a data processing method based on the Structure-Preserving and Semantically-Reordered Code Graph for constructing model training corpora, significantly enhancing the model's perception and utilization of cross-file, cross-module code structures and semantic dependencies. The effectiveness and superiority of the method are validated in the communication domain's "on-the-fly completion" task.

2 Related Work

2.1 Evaluation Metrics for Code Completion Tasks

Currently, many evaluation metrics are used for code completion tasks. Common metrics include Exact Match (EM), Unit Testing (UT), Edit Similarity (ES), BLEU, ROUGE-L, and ROUGE-N (Zhuo et al., 2024; Ding et al., 2023; Wang et al., 2024; Izadi et al., 2024; Yang et al., 2024; Austin et al., 2021). Although unit testing is considered the gold standard for code tasks, its high production cost makes it impractical for large-scale use. For code completion tasks, researchers tend to use EM and ROUGE-L due to the availability of context and ground truth (Li et al., 2025; Jiang et al., 2025; Izadi et al., 2024). However, in our practice, we find that EM and ROUGE-L still fall short in capturing user perception for "on-the-fly completion" tasks.

2.2 Repository-Level Code Completion

Repository-level code completion scenarios often require long-context inputs spanning multiple files and modules, inevitably introducing redundant information and reducing generation quality (Shi et al., 2023). To address this, CoCoMIC (Ding et al., 2024) and RepoFusion (Shrivastava et al., 2023) enhance models' context-awareness through long-context fine-tuning of LLMs. To further reduce contextual redundancy, CodeT5 (Wang et al., 2021) and PLBART (Ahmad et al., 2021) construct hard-structured code corpora using abstract syntax trees (ASTs); AST-T5 (Gong et al., 2024) further constructs soft-structured code corpora with high model affinity using ASTs, improving the model's perception capabilities. GraphCoder (Liu et al., 2024b) introduces control flow graphs to build Code Context Graphs and injects context through Retrieval-Augmented Generation (RAG), improving the model's accuracy in repository-level code completion tasks.

However, existing models still suffer from insufficient structural awareness when handling long-context inputs spanning multiple files and modules.

3 Method

3.1 User Behavior Analysis

Definition of Adoption: Developers accept the code suggestions provided by Copilot and directly apply them to the project, either as-is or after modi-

fication. Here, we consider each press of the "Tab" key by the user as one adoption.

Definition of Adoption Rate: (Number of adopted code suggestions / Total number of code suggestions generated by Copilot during development) * 100%

User Behavior Analysis: From a practical perspective, in code completion scenarios, the higher the similarity between the model's output and the reference answer, the more likely users are to adopt it. However, considering the left-to-right code editing habits of users, they often pay special attention to whether the model's output starts correctly from the first character. Even if only part of the output is correct, users may still adopt it and manually modify the result.

Therefore, we can reasonably infer that the longer the longest common prefix (LCP) between the model's output and the reference answer, the higher the user adoption rate.

This observation suggests that traditional metrics such as Exact Match (EM) or ROUGE-L may not fully reflect the model's performance in real-world usage scenarios. Instead, metrics focusing on continuous prefix matching better align with user expectations and interaction patterns.

To address this issue, we propose two new evaluation metrics that better align with user perception:

- Longest Common Prefix (LCP);
- A variant of ROUGE-L —— ROUGE-LCP.

Next, we will analyze the theoretical foundation of these proposed metrics based on the probabilistic modeling mechanism of large language models (LLMs) and establish their probability distribution models to explain the relationship between these metrics and user adoption behavior.

3.2 Design of LCP and ROUGE-LCP Metrics

Longest Common Prefix (LCP): The Longest Common Prefix (LCP(S, R)) is defined as the maximum number of consecutive characters that match between the model's output S and the reference text R, starting from the beginning.

This metric emphasizes continuity starting from the first character, which is a critical feature in interactive code writing and editing. Unlike LCS-based metrics that allow non-contiguous matches, LCP aligns more closely with real user experiences.

Based on the probability formula for LLM outputs, we can model the correctness probability of the first n characters of the model's output, i.e., derive the probability distribution of LCP.

Let n represent the number of consecutive correct characters before the first incorrect position. Let the model's output sequence be $S = s_1, s_2, ..., s_T$, and the reference sequence (i.e., the actual code) be $R = r_1, r_2, ..., r_T$. In other words, $n = k$ means the first k characters are correct (i.e., $s_1 = r_1, ..., s_k = r_k$), while the $(k+1)$-th character is the first error (i.e., $s_{k+1} \neq r_{k+1}$).

The probability of $n = k$ is modeled as:

$$P(n = k) = P(LCP(S, R) = k) = \left(\prod_{t=1}^{k} P(s_t = r_t \mid s_1 = r_1, \ldots, s_{t-1} = r_{t-1})\right) \cdot (1 - P(s_{k+1} = r_{k+1} \mid s_1 = r_1, \ldots, s_k = r_k))$$

This is a long-tail distribution, reflecting the increasing difficulty of maintaining correctness over longer sequences.

This metric correlates well with user adoption rates and better reflects the model's actual capabilities in code completion. Moreover, the probability distribution of this metric aligns with the correctness distribution of the model's output, making it compatible with loss functions and providing clear guidance for model training. Unlike LCS-based metrics that allow non-contiguous matches, our definition emphasizes continuity starting from the first character — a key aspect of interactive code writing and editing.

ROUGE-LCP: Inspired by the normalization concept of ROUGE-L, we further propose a normalized evaluation metric based on LCP, defined as follows:

$$\text{ROUGE-LCP}(S, R) = \frac{\text{LCP}(S, R)}{|R|}$$

This metric divides the LCP value by the length of the reference sequence $|R|$, enabling fair comparisons across samples of different lengths. Since LCP strongly correlates with user adoption behavior, we have reason to believe that ROUGE-LCP can also effectively reflect the practicality of model outputs in interactive scenarios. Experimental results also validate this hypothesis. To further understand the statistical properties of ROUGE-LCP and establish a mathematical connection with Exact Match (EM), we introduce an auxiliary parameter $|S_{\text{ext}}|$, representing the length of the portion of the model's output that extends beyond the reference sequence when $\text{LCP}(S, R) = |R|$. Based on this, we can model the probability distribution of ROUGE-LCP as a mixed distribution, as follows:

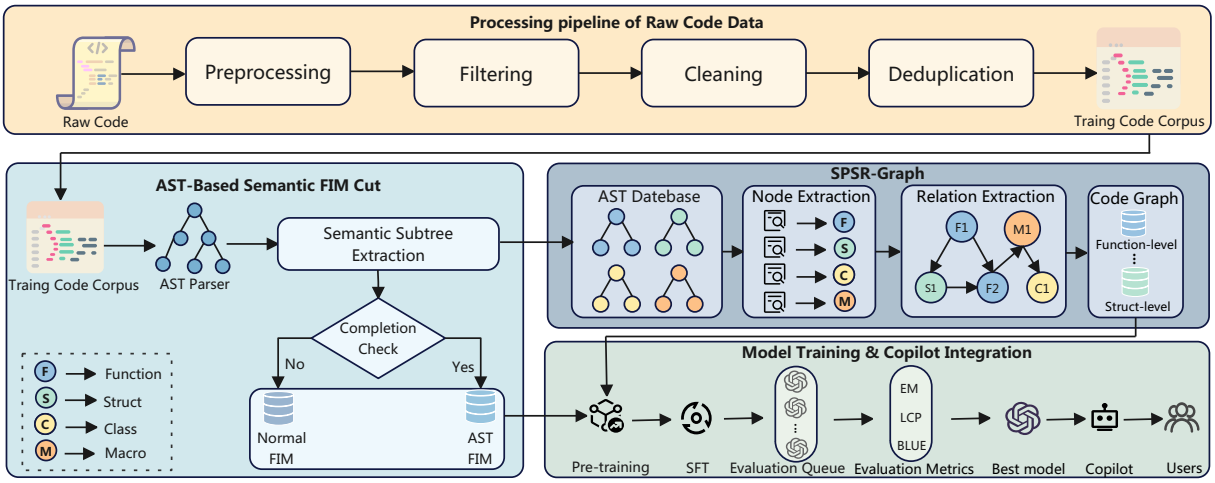

Figure 1: Overview of the proposed framework. The framework includes three stages: corpus preprocessing, AST structural segmentation, and structure-aware graph construction, transitioning from local semantic preservation to global dependency modeling to systematically construct structure-aware code completion corpora.

$$P(\text{Rouge-LCP}(S,R)) = \begin{cases} P\left(\dfrac{\text{LCP}(S,R)}{|R|}\right), & \text{if LCP}(R,S) < |R|; \\ P(\text{EM}(S,R)), & \text{if LCP}(R,S) = |R| \; and \; S = R; \\ P\left(\dfrac{\text{LCP}(S,R) + |S_{\text{ext}}|}{|R|}\right), & \\ \qquad \text{if LCP}(R,S) = |R| \; and \; S \neq R; & \end{cases}$$

According to the Central Limit Theorem, the length of the reference text $|R|$ typically follows an approximately Gaussian distribution in real-world settings. This formula reveals the dual nature of ROUGE-LCP:

When the output prefix does not fully match the reference text, the metric value is less than 1, corresponding to partially correct cases; When the output prefix fully matches the reference text, the metric value is 1, and the output may continue generating additional content (i.e., $|S_{\text{ext}}| > 0$), which should also be included in the evaluation scope.

Thus, the distribution of ROUGE-LCP can be viewed as the superposition of two sub-distributions:

Partial Match Component: Captures the model's ability to generate partially correct prefixes;

Exact Match Component: Measures the model's ability to not only generate correct prefixes but also continue producing useful content beyond that.

This modeling approach allows us to analyze the model's performance at different stages within a unified framework and provides a theoretical basis for further exploring its relationship with user adoption behavior. In subsequent experiments, we will separately analyze the correlation between these two components and user adoption rates.

3.3 Repository-Level Code Corpus Processing Framework

In this section, we describe in detail the corpus preprocessing pipeline, AST-based semantic segmentation, and the construction process of the code knowledge graph. Corpus preprocessing aims to improve the quality of training corpora by reducing noise and irrelevant information; AST-based semantic segmentation ensures the integrity of semantic units and prevents semantic disruption; while the code knowledge graph focuses on further constructing and enhancing the relationships between semantic units, thereby improving the model's global and cross-library semantic understanding capabilities.

To ensure that the generated knowledge graph corpora are of high quality, we have established a systematic preprocessing workflow, including key steps such as data filtering, data cleaning, and deduplication. Specific filtering rules, cleaning methods, and deduplication processes are detailed in Appx.A.

3.3.1 Syntax-Aware Semantic Unit Extraction via AST

To enhance the structural awareness of pretraining corpora, we propose an AST-based semantic segmentation method as an alternative to traditional random or sliding-window masking strategies based on tokens. This method uses semantically closed subtrees in the AST as segmenta-

tion units, ensuring the structural integrity and contextual continuity of the masked units. Specifically, the method includes the following four steps: First, tools like Tree-sitter are used to parse the source code and extract semantically complete AST subtrees, such as function bodies and conditional branches. Second, a subtree is randomly sampled as the masking target, replaced with a placeholder, and concatenated with its preceding and succeeding contexts to form the training input. Third, structural integrity checks are performed to ensure that the masking operation does not disrupt the syntactic parsability of the remaining code. Finally, a granularity control parameter θ is introduced to adjust the size of the subtrees, supporting multi-scale structural modeling. This method can be completed in linear complexity and is suitable for large-scale code corpus construction. For the formal modeling, algorithm, and comparative analysis with greedy segmentation, please refer to Appx.B.

3.3.2 Structure-Preserving and Semantically-Reordered Code Graph

After AST-based semantic unit extraction, segmentation, and completeness verification, we build a Structure-Preserving and Semantically-Reordered Code Graph (SPSR-Graph) to generate training corpora that maintain global call consistency. The construction proceeds in two stages: (1) semantic-unit extraction, where we parse the vertical-domain codebase to obtain self-contained units such as functions, structs, and classes; and (2) semantic-relationship graphing, where we connect these units with directed edges that encode calls, references, and inclusions. Traversing this graph along call paths allows us to reorder source code into contextually aligned sequences, enriching the structural depth of the corpus and enabling repository-level, cross-library context modeling.

To further extend the structural depth of training corpora and enhance cross-library context modeling, we propose organizing semantic units into structured graphs, constructing a semantic dependency graph named SPSR-Graph. The graph construction process is divided into two stages: semantic unit extraction and semantic relationship graphing. The SPSR-Graph construction process consists of two stages:

The first stage is element extraction. We use an AST parser to parse the entire codebase and extract all top-level semantic units $\nu_i \in \boldsymbol{\nu}$, such as function bodies, structs, and class definitions. Each ν_i is semantically complete and stored in a structured database for subsequent calls.

The second stage is relationship extraction and graph construction. We represent the code graph as $\mathbf{\Gamma} = (\mathcal{V}, \epsilon)$, where $\mathcal{V}$ is the set of nodes, i.e., the extracted semantic units; $\epsilon \subseteq \mathcal{V} \times \mathcal{V}$ is the set of directed edges. If there is a call relationship $\nu_i \rightarrow \nu_j$, we define the edge $(\nu_i, \nu_j) \in \epsilon$. Edge types can support, but are not limited to: direct function calls (Direct Call), member references (Member Reference), type dependencies (Type Usage), macro or template expansions (Macro Expansion), and file inclusions (Include Dependency). To preserve contextual integrity, graph construction supports node attribute enhancement (e.g., definition location, module affiliation, syntax type labels) and edge type annotations, further enhancing the graph's semantic capacity.

On the directed graph $\mathbf{\Gamma}$, we use directed BFS to search for all paths $\mathcal{P}$ with depth $d \leq D$:

$$\mathcal{P} = \{p_k = (\nu_{k_1}, \nu_{k_2}, ..., \nu_{k_m}) \mid \nu_{k_i} \in \mathcal{V},\ m \leq D\}$$

Path selection supports multiple strategies: forward call expansion (Forward Call Expansion), field access chain expansion (Field Access Expansion), and header inclusion prioritization (Header Inclusion). Each path p_k is mapped to the following training sample:

$$\text{Sample}(p_k) = \nu_{k_1} \oplus \nu_{k_2} \oplus \ldots \oplus \nu_{k_m}$$

where $\oplus$ denotes structure-aware concatenation.

To enhance the model's cross-file structural modeling capability, we insert file path information and structural comments during concatenation:

$$\nu_{k_i} \mapsto \texttt{/* file: path/to/file */} \oplus \text{code}$$

The following algorithm explicitly shows the training sample construction process for SPSR-Graph. First, the system constructs the graph structure using the extracted AST semantic units, where each node represents a semantically complete code unit, and edges represent call or reference relationships. Then, breadth-first traversal (BFS) is used to enumerate all semantic paths with depth not exceeding D.

During traversal, for each valid path in the graph, the algorithm sequentially loads the source code fragments corresponding to each node in the path and embeds structural annotation information at cross-file boundaries. Finally, the structured fragments contained in the entire path are concatenated in dependency order to form a training sample with global semantic consistency, which is stored for

subsequent language model pretraining. Let the total number of nodes be n, the average outdegree be d, and the maximum path depth be D. Then, the complexity is: $\mathcal{O}(n + nd + n \cdot d^D \cdot m)$.

This process preserves both syntactic structural integrity and contextual consistency while achieving corpus reordering along call paths, enabling the model to explicitly encounter and model cross-function and cross-module structural dependencies during training.

Algorithm 1: SPSR-Graph Generator

Input: AST-unit DB `ASTDB`; max depth D
Output: structure–aware sample set $\mathcal{S}$

$\Gamma \leftarrow$ INITGRAPH() **for each** $u \in$ `ASTDB`
  Γ.addNode(u) **for each**
  $(u, v) \in$ CALLPAIRS(`ASTDB`) Γ.addEdge($u \rightarrow v$)
  $\mathcal{S} \leftarrow \emptyset$;
for each $p \in$ `BFSPaths`(Γ, D)
  $s \leftarrow$ CONCAT(`Tag`(u) $\mid u \in p$);
$\mathcal{S} \leftarrow \mathcal{S} \cup \{s\}$;
return $\mathcal{S}$

Function `BFSPaths`(Γ, D)
  return *all directed paths of* Γ *with length* $\leq D$ *(BFS)*

Function `Tag`(u)
  if CROSSFILE(u) **then**
    return `/* file:` *u.file* `*/` $\|$ *code(u)*
  else
    return *code(u)*

4 Experiments

4.1 Experimental Setup

This experiment uses Qwen2.5-7B-Coder as the base model (Hui et al., 2024), adopting a 28-layer Transformer architecture with a total of approximately 7.6 billion parameters. The model is pretrained on a 0.6B C/C++ code corpus from the communication domain and fine-tuned on approximately 60,000 lines of code corpus from the same domain. The fine-tuning corpus includes fields such as context, the line to be completed, and similar code retrieved via RAG. Model training was conducted on a single server equipped with 8 NVIDIA A100 80GB GPUs. We evaluated model performance using EM, LCP, and BLEU on user data.

We collected on-the-fly completion data from March 3, 2025, to April 24, 2025, using data logging for evaluation and analysis. The data source is XXX-Code-Copilot. The logged information includes timestamps, trigger points, programming languages, model predictions, context, and reference answers (i.e., the content confirmed by the user pressing the Enter key).

During the data preprocessing stage, we filtered out duplicate samples and data where the context contradicted the reference answers, resulting in a final dataset of 10,769 valid entries. Among these, 6,100 entries were from March 3 to March 31, and 4,669 entries were from April 1 to April 24. We divided this data into three groups for analysis: the first group covers March 3 to March 31; the second group covers April 1 to April 24; the third group spans the entire period.

4.2 LCP and ROUGE-LCP Distribution and Correlation with Adoption Rate

We first analyze the distribution of LCP and its correlation with the adoption rate in the three datasets.

As shown in Fig. 2, the distribution of LCP has a clear long-tail property, which aligns well with the probabilistic structure derived in our theoretical modeling. Furthermore, we calculated the Pearson correlation coefficient (Pearson's r) between LCP and the adoption rate and found a significant positive correlation. As shown in Table 1, across all time periods, the r values were above 0.6, and the P values were below 0.05, fully validating the effectiveness of the LCP metric.

Additionally, when LCP = 1, the adoption rate shows a local peak. This phenomenon can be attributed to users often adopting punctuation marks or syntactic structures such as ","":","";" during code completion. Although these elements carry low information content, they still contribute to improving coding efficiency.

Next, we analyze the relationship between ROUGE-LCP counts, adoption counts, and adoption rates. Specifically, when $\text{LCP}(S, R) = |R|$, we split this scenario into two parts: the exact match (EM = 1) component and the partial extension component represented by $\frac{\text{LCP}(S,R)+|S_{\text{ext}}|}{|R|}$, evaluating their respective relationships with user adoption rates.

As shown in Fig. 3, the results indicate that the distribution of ROUGE-LCP follows a mixed distribution, consistent with our expectations. As the ROUGE-LCP value increases, the model's adoption rate generally rises, peaking at EM = 1. Even when the model output exceeds the reference sequence length (i.e., $|S_{\text{ext}}| > 0$), there is still a certain level of adoption rate, suggesting that users tend to adopt and modify partially correct outputs. More results can be found in Appx.D

Metric	0303-0331	0401-0424	0303-0424
r	0.9107	0.6952	0.8707
P	0.0000	0.0028	0.0000

Table 1: Pearson Correlation Analysis Between LCP and Adoption Rate

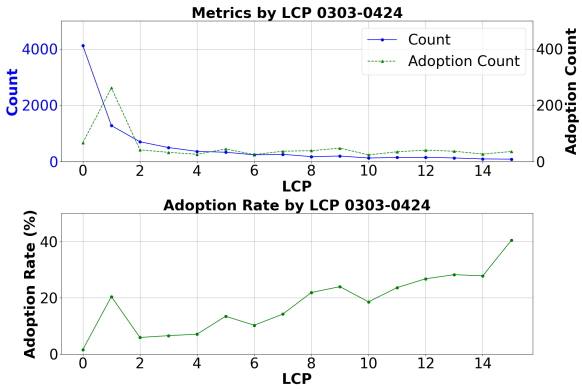

Figure 2: LCP Distribution and Its Relationship with Adoption Count and Adoption Rate

4.3 Comparison with General Metrics

To improve statistical stability, we filtered out data points with fewer than 100 completions per day, as these samples had low adoption counts, which could lead to significant fluctuations in adoption rates. Subsequently, we calculated the correlations between LCP, LCS, ROUGE-LCP, ROUGE-L, EM, and adoption rate on a daily basis to verify the effectiveness of the proposed metrics in reflecting user perception.

As shown in Fig. 4 and Table 2, by observing the correlation heatmap, we found that compared to commonly used code completion metrics, LCP showed the strongest correlation with the adoption rate, with r values generally exceeding 0.7 and P values below 0.05; ROUGE-LCP followed closely. These results further validate that the proposed LCP and ROUGE-LCP metrics are better at capturing user intent and adoption behavior than general-purpose metrics. More results can be found in Appx.C

Metric	0303-0331	0401-0424	0303-0424	
		r		P<0.05
LCP	**0.7145**	**0.7862**	**0.7485**	✓
ROUGE-LCP	0.6224	0.7186	0.6867	✓
LCS	0.5863	0.7184	0.6677	✓
ROUGE-L	0.5729	0.6914	0.6498	✓
EM	0.5917	0.7037	0.6464	✓

Table 2: Pearson Correlation Analysis Between Common Evaluation Metrics (LCS, LCP, ROUGE-L, ROUGE-LCP, EM) and Adoption Rate

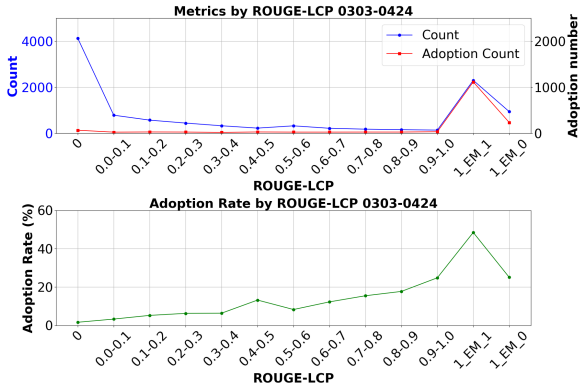

Figure 3: ROUGE-LCP Distribution and Its Relationship with Adoption Count and Adoption Rate

4.4 Impact of Different Pretraining Corpus Strategies

This section sets up four groups of pretraining corpora for model pretraining, followed by fine-tuning using the same fine-tuning corpus described above, aiming to evaluate the impact of the proposed structure-aware corpus processing strategy on the "on-the-fly completion" task.

Pipeline: Applying a data filtering, cleaning, and deduplication pipeline to the raw corpus;

AST: Applying AST-based semantic segmentation on top of the pipeline-processed corpus;

KG_F: Constructing function-level code graph corpora based on AST-based semantic segmentation;

KG_{FS}: Further introducing struct-level code graph corpora.

	C++			C		
Corpus	**EM(%)**	**LCP**	**Blue**	**EM(%)**	**LCP**	**Blue**
Pipeline	16.54	5.2	29.33	17.31	5.1	29.21
+ AST	16.72	4.8	27.83	17.70	5.1	28.90
+ KG_F	17.84	5.2	29.47	20.36	**5.8**	31.64
+ KG_{FS}	**18.55**	**5.2**	**29.73**	**20.51**	5.7	**31.79**

Table 3: Performance comparison of different pretraining corpus strategies on the "on-the-fly completion" task

As shown in Table 3, first, replacing random FIM with semantically complete AST-based segmentation further improves EM, demonstrating the benefit of consistent semantic unit boundaries in enhancing the model's perception of completion scope. Although BLEU slightly decreases, LCP remains stable, indicating that structural segmentation has no negative impact on prefix prediction.

Furthermore, the function-level code knowledge graph brings significant improvements, especially in C, where EM increases by 2.66% and BLEU

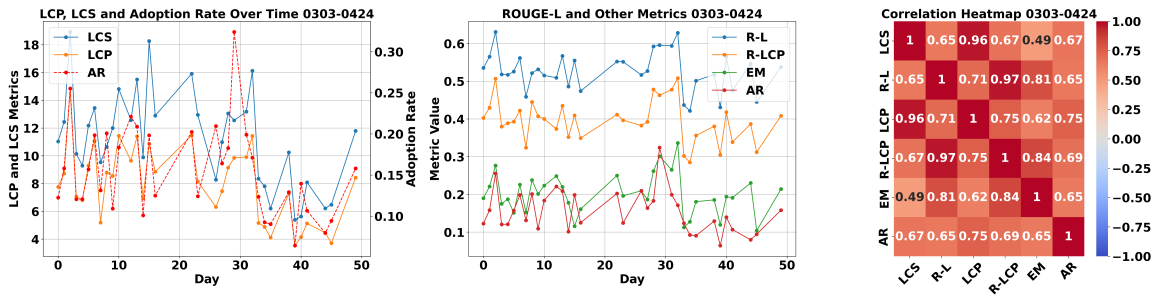

Figure 4: Daily Metric and Adoption Rate Distributions, Heatmap of Correlation Between Evaluation Metrics and Adoption Rate. R-L refers to ROUGE-L, R-LCP refers to ROUGE-LCP, and AR refers to Adoption Rate.

by 2.74 compared to AST alone. This suggests that after semantic reordering at the graph level, real cross-library information can be introduced, enabling the model to better capture global contextual relationships. Finally, adding struct-level knowledge graphs brings additional gains, particularly in complex languages like C++, where edge augmentation helps model the cross-node impact of type information.

In summary, the 3-stage evolution strategy—from data cleaning, structure-aware segmentation, to graph-based semantic reordering-improves the structural integrity and dependency consistency of the corpus, yielding consistent gains across multiple language tasks. The dual improvement in EM and BLEU, coupled with the steady growth in LCP, validates the practical effectiveness of structure-aware training samples.

4.5 Impact of Code Knowledge Graph Breadth

In this section, we explore the impact of graph traversal breadth on model performance. Since this task focuses on single-line or inline completion, which typically involves only local context, deep dependency modeling is unnecessary; thus, the traversal depth is fixed at 1. Considering the average dependency count of 3–5 in codebases and the model's maximum input length, we select $k = 3, 4, 5, 6, 7$ as the maximum breadth limit during graph construction.

As shown in Figure 5, the breadth of the knowledge graph has a non-monotonic impact on model completion performance: when the breadth increases from 3 to 4, all metrics for both C and C++ improve significantly, reaching their peak; however, further increasing the breadth to 5–7 results in a slight decline or stabilization in performance. When the graph breadth becomes too large, it may introduce irrelevant context, distracting the model's attention and reducing completion accuracy;

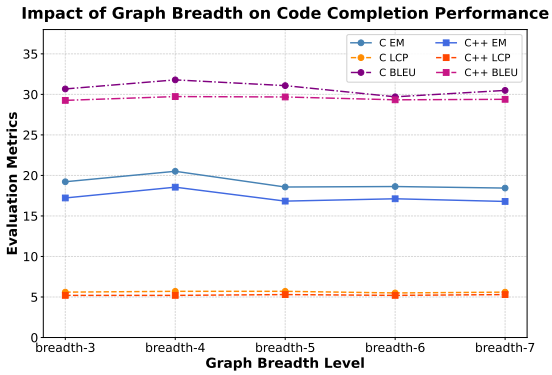

Figure 5: Impact of code knowledge graph breadth on code completion performance

Additionally, compared to C++, C shows greater sensitivity to changes in graph breadth, indicating that dependencies between functions in C are more concentrated, benefiting less from hints beyond the local scope. In contrast, C++'s more complex structure makes it less sensitive to edge expansion, reflecting its stronger adaptability to type references.

5 Conclusion

This paper proposes two evaluation metrics for the "on-the-fly completion" task: LCP and ROUGE-LCP. We theoretically analyze the relationship between these metrics and the model training objectives. Based on real user behavior data, we compare these metrics with other general evaluation metrics and validate the effectiveness of the proposed metrics in measuring user adoption rates. Additionally, we proposes a data processing method based on a Structure-Preserving and Semantically-Reordered Code Graph. Experimental validation demonstrates that this method significantly enhances the model's ability to perceive code structure and semantic dependencies, thereby effectively improving the model's performance in practical tasks.

6 Limitations

This paper focuses only on on-the-fly completion tasks and provides theoretical analysis and empirical validation for the proposed metrics. The current metrics in terms of model affinity are applicable only to autoregressive generative models, and their adaptability to other types of models remains to be further studied. The results indicate that when deploying large language models in real-world production environments, it is essential to design evaluation metrics tailored to the model's characteristics and business scenarios, in order to more accurately assess the model's practical performance and user experience.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 2655–2668. Association for Computational Linguistics.

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

Gareth Ari Aye, Seohyun Kim, and Hongyu Li. 2021. Learning autocompletion from real-world datasets. In *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '21, page 131–139. IEEE Press.

Vitaliy Bibaev, Alexey Kalina, Vadim Lomshakov, Yaroslav Golubev, Alexander Bezzubov, Nikita Povarov, and Timofey Bryksin. 2022. All you need is logs: improving code completion by learning from anonymous ide usage logs. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 1269–1279, New York, NY, USA. Association for Computing Machinery.

Cursor. 2023. https://www.cursor.com.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2024. Cocomic: Code completion by jointly modeling in-file and cross-file context. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation, LREC/COLING 2024, 20-25 May, 2024, Torino, Italy*, pages 3433–3445. ELRA and ICCL.

GitHub. 2021. https://github.com/features/copilot.

Linyuan Gong, Mostafa Elhoushi, and Alvin Cheung. 2024. AST-T5: structure-aware pretraining for code generation and understanding. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

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, and 5 others. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Maliheh Izadi, Jonathan Katzy, Tim Van Dam, Marc Otten, Razvan Mihai Popescu, and Arie Van Deursen. 2024. Language models for code completion: A practical evaluation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

Siyuan Jiang, Jia Li, He Zong, Huanyu Liu, Hao Zhu, Shukai Hu, Erlu Li, Jiazheng Ding, Yu Han, Wei Ning, Gen Wang, Yihong Dong, Kechi Zhang, and Ge Li. 2025. aixcoder-7b: A lightweight and effective large language model for code processing. *Preprint*, arXiv:2410.13187.

Jia Li, Hao Zhu, Huanyu Liu, Xianjie Shi, He Zong, Yihong Dong, Kechi Zhang, Siyuan Jiang, Zhi Jin, and Ge Li. 2025. aixcoder-7b-v2: Training llms to fully utilize the long context in repository-level code completion. *Preprint*, arXiv:2503.15301.

Junwei Liu, Yixuan Chen, Mingwei Liu, Xin Peng, and Yiling Lou. 2024a. Stall+: Boosting llm-based repository-level code completion with static analysis. *arXiv preprint arXiv:2406.10018*.

Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. 2024b. Graphcoder: Enhancing repository-level code completion via code context graph-based retrieval and language model. *CoRR*, abs/2406.07003.

Mengnan Qi, Yufan Huang, Yongqiang Yao, Maoquan Wang, Bin Gu, and Neel Sundaresan. 2024. Is next token prediction sufficient for gpt? exploration on code logic comprehension. *Preprint*, arXiv:2404.08885.

Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H. Chi, Nathanael Schärli, and Denny Zhou. 2023. Large language models can

be easily distracted by irrelevant context. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 31210–31227. PMLR.

Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. 2023. Repofusion: Training code models to understand your repository. *CoRR*, abs/2306.10998.

Wannita Takerngsaksiri, Chakkrit Tantithamthavorn, and Yuan-Fang Li. 2024. Syntax-aware on-the-fly code completion. *Inf. Softw. Technol.*, 165(C).

Tim van Dam, Maliheh Izadi, and Arie van Deursen. 2023. Enriching source code with contextual data for code completion models: An empirical study. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 170–182.

Tim Van Dam, Frank Van der Heijden, Philippe De Bekker, Berend Nieuwschepen, Marc Otten, and Maliheh Izadi. 2024. Investigating the performance of language models for completing code in functional programming languages: a haskell case study. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, pages 91–102.

Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024. Rlcoder: Reinforcement learning for repository-level code completion. *Preprint*, arXiv:2407.19487.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics.

Qinyun Wu, Chao Peng, Pengfei Gao, Ruida Hu, Haoyu Gan, Bo Jiang, Jinhe Tang, Zhiwen Deng, Zhanming Guan, Cuiyun Gao, and 1 others. 2024. Repomastereval: Evaluating code completion via real-world repositories. *arXiv preprint arXiv:2408.03519*.

Jian Yang, Jiaxi Yang, Ke Jin, Yibo Miao, Lei Zhang, Liqun Yang, Zeyu Cui, Yichang Zhang, Binyuan Hui, and Junyang Lin. 2024. Evaluating and aligning codellms on human preference. *Preprint*, arXiv:2412.05210.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A Corpus Preprocessing Details

This appendix supplements the detailed implementation of the corpus preprocessing pipeline described in the methodology section of the main text, including data filtering, data cleaning, and deduplication.

A.1 Data Filtering

The data filtering stage aims to remove code samples that do not meet training standards. The specific rules include:

Line Length Threshold Filtering: Remove code samples where any line length falls outside the specified range to avoid abnormal or excessively short code.

Average Line Length Filtering: Remove samples with an average line length exceeding the threshold to ensure balanced code structure.

Character Validity Ratio Filtering: Discard samples with a low proportion of alphanumeric characters to exclude non-programmatic text.

Total Character Count Filtering: Delete files with a total character count below the minimum threshold to prevent fragmented code from entering the training set.

File Type Filtering: Explicitly exclude non-target language files (e.g., XML, HTML) to ensure the purity and specificity of the training corpus.

A.2 Data Cleaning

To improve corpus consistency and semantic validity, we introduce the following cleaning strategies:

Code Formatting Standardization: Use automated formatting tools to unify code style, enhancing consistency and parsability.

Comment and Whitespace Cleanup: Remove invalid comments, redundant spaces, and blank lines to minimize interference and avoid negative impacts on the model.

A.3 Data Deduplication

To prevent duplicate code samples from affecting training efficiency and model generalization, we adopt a two-stage deduplication strategy:

Exact Deduplication: Compute the SHA256 hash value for each code sample and remove completely identical samples.

Fuzzy Deduplication: Use the MinHash algorithm to detect structural and semantic similarity between samples, further removing highly redundant code fragments. This strategy not only reduces

redundant data but also effectively preserves corpus diversity.

B Syntax-Aware Semantic Unit Extraction via AST

This appendix supplements the complete implementation and theoretical analysis details of the AST-based structure-aware segmentation method described in the methodology section of the main text. Specifically, it includes: AST syntax modeling and semantic unit definition, pseudocode implementation of the AST semantic segmentation algorithm, and a theoretical analysis of traditional greedy segmentation.

Fill-in-the-Middle (FIM) is a widely used pretraining strategy for code completion tasks, where the core idea is to mask code segments and let the model predict the masked content. However, traditional FIM samples are often based on token-level greedy segmentation, such as fixed-step sequential partitioning or random cutting. While this approach is simple to implement, it can easily disrupt semantic structures in programs, such as functions, conditional branches, and loop bodies, thus limiting the model's ability to model structural information. To address this, we propose an Abstract Syntax Tree (AST)-based semantic-aware segmentation method to enhance the consistency and integrity of structural units in training samples.

B.1 AST Modeling and Semantic Unit Definition

An abstract syntax tree is a structured representation of source code in terms of syntax. Formally, we represent an AST as an ordered tree $T = (V, E)$. Here, $V = \{v_i\}_{i=1}^{n}$ is the set of nodes representing syntactic units such as keywords, expressions, and statement blocks; $E \subseteq V \times V$ is the set of edges, with each edge (v_i, v_j) indicating that v_j is a syntactic child node of v_i, forming a hierarchical structure recursively expanded from parent nodes based on grammar rules. Each node v_i has a label t_i:

$$\text{label}(v_i) = t_i, \quad t_i \in \mathcal{T}$$

where $\mathcal{T}$ is the set of non-terminal symbols defined by the language, such as `FunctionDefinition`.

Given a node $v_i \in V$, we define the AST subtree rooted at v_i as $T_s(v_i) \subset T$. This subtree contains all child nodes reachable from v_i via the set of syntactic edges E. We refer to $T_s(v_i)$ as a "semantically complete unit," as it forms a closed, independently modelable structural fragment in terms of both syntax and semantics, such as a function body or control flow block.

B.2 Overview of AST Semantic Segmentation Method

We propose a structure-aware training sample construction method based on abstract syntax trees, aiming to replace the traditional FIM sample generation strategy based on linear token segmentation. This method follows four key steps:

1. **Semantic Unit Extraction**: Using syntax parsing tools like Tree-sitter, extract semantically closed AST subtrees from the source code. Each subtree $T_s(v_i)$ represents a structurally modelable code unit (e.g., function definition, conditional branch, or loop body) with complete syntactic boundaries and internal semantic consistency.

2. **Structure-Consistent Mask Construction**: Randomly sample a semantic unit $S = T_s(v_i)$ from the candidate subtrees, constructing the input as:

   $$\text{Input} = P_1 \texttt{<mask>} P_2, \quad \text{Target} = S$$

   where P_1 and P_2 are the preceding and succeeding contexts of the subtree in the original source code, respectively. This design ensures that the generated training samples remain consistent with real-world code structures.

3. **Semantic Completeness Verification**: After each masking operation, verify whether the remaining AST still constitutes a syntactically valid tree. If the masking disrupts the closure of the AST (e.g., breaking control statements or expression blocks), discard the candidate subtree to ensure the structural correctness of the training samples.

4. **Structural Granularity Control Mechanism**: Introduce a masking granularity control parameter θ to adjust the size of the masked subtree (e.g., number of tokens or nodes). By sampling $\theta \in [\theta_{\min}, \theta_{\max}]$, this method supports multi-scale semantic modeling, ranging from micro-level expression masking to macro-level function masking.

This method can be formalized as the following algorithm. We first parse the source code to construct the corresponding abstract syntax tree (AST),

then traverse its structural semantic units. Under the constraints of granularity and structural completeness, we construct standard Fill-in-the-Middle training samples for each candidate subtree. The complete process is as follows:

Algorithm 2: AST Semantic FIM Cut

Input: Code file C; threshold θ
Output: A set of structure-aligned FIM training samples

$AST \leftarrow$ Parse(C);
$Nodes \leftarrow$ ExtractSemanticSubtrees(AST);
$Masked \leftarrow \emptyset$;
for each $T_s \in Nodes$ **if** Size(T_s) $\leq \theta$ ***and*** IsComplete($AST \setminus T_s$) **then**
  $(P_1, P_2) \leftarrow$ ContextAround(T_s);
  $Sample \leftarrow$ {"input" : $P_1 + \langle \texttt{MASK} \rangle + P_2$, "target" : T_s};
  Masked.append($Sample$);
return $Masked$

We use depth-first traversal (DFS) to extract subtrees from the abstract syntax tree, with a time complexity of $\mathcal{O}(|V|)$. The completeness check of the masked region can also be completed in linear time, with the same complexity of $\mathcal{O}(|V|)$. Therefore, the overall segmentation process incurs linear overhead during the preprocessing stage, making it suitable for large-scale code pretraining tasks.

B.3 Theoretical Analysis: Greedy Segmentation vs. Structural Disruption

In traditional FIM task construction, samples are often generated using a greedy segmentation strategy, which divides the original code sequence into segments of fixed token window length ℓ. Assuming a function body contains L total tokens, it will be divided into $k = \lceil L/\ell \rceil$ segments.

Since greedy segmentation does not perceive any syntactic structure, its segmentation boundaries may fall within semantic units such as function bodies, conditional statements, or loop bodies, causing semantic fragmentation. We define the structural preservation rate (the probability that a semantic unit remains unbroken) as:

$$R_s = \frac{1}{k}$$

while the remaining $k-1$ segments are structurally disrupted, with a proportion of $1 - \frac{1}{k}$. As k increases, the structural disruption rate rises sharply, especially when dealing with large functions or deeply nested structures, significantly interfering with the model's ability to model program structures.

In contrast, AST-based semantic segmentation uses syntactic structures as boundaries, ensuring that segmentation units are complete semantic subtrees. This method theoretically guarantees that no semantic structure is broken, with a structural preservation rate of $R_s = 1.0$. Thus, in structural modeling tasks, the AST segmentation strategy provides significantly stronger structural consistency guarantees.

C LCP and ROUGE-LCP Distribution and Correlation with Adoption Rate in all time periods

Here, we present the LCP and ROUGE-LCP distributions and their correlation with the adoption rate for the periods from March 3 to March 31, April 1, and April 24. As shown in Fig. 6 and Fig. 7. The three datasets consistently exhibit the same characteristics: the distribution of LCP shows a significant long-tail property, which aligns well with the probability structure derived in our theoretical modeling. From the figures, it can be observed that there is a significant positive correlation between LCP and the adoption rate. Additionally, when LCP equals 1, the adoption rates of all three datasets show a local peak. Next, we examine the relationship between the count, adoption frequency, and adoption rate of ROUGE-LCP across the three time periods. The results indicate that the distribution of ROUGE-LCP exhibits a mixed distribution, consistent with our expectations. As the ROUGE-LCP value increases, the model's adoption rate generally rises and reaches its highest point when EM = 1.

D LCP and ROUGE-LCP Comparison with General Metrics in all time periods

Here, we also present the Daily Metric and Adoption Rate Distributions, as well as the Heatmap of Correlation Between Evaluation Metrics and Adoption Rate for the periods from March 3 to March 31, April 1, and April 24. As shown in Fig. 8 We calculate the correlations between LCP, LCS, ROUGE-LCP, ROUGE-L, EM, and the adoption rate on a daily granularity to verify the effectiveness of the proposed metrics in reflecting user perception.

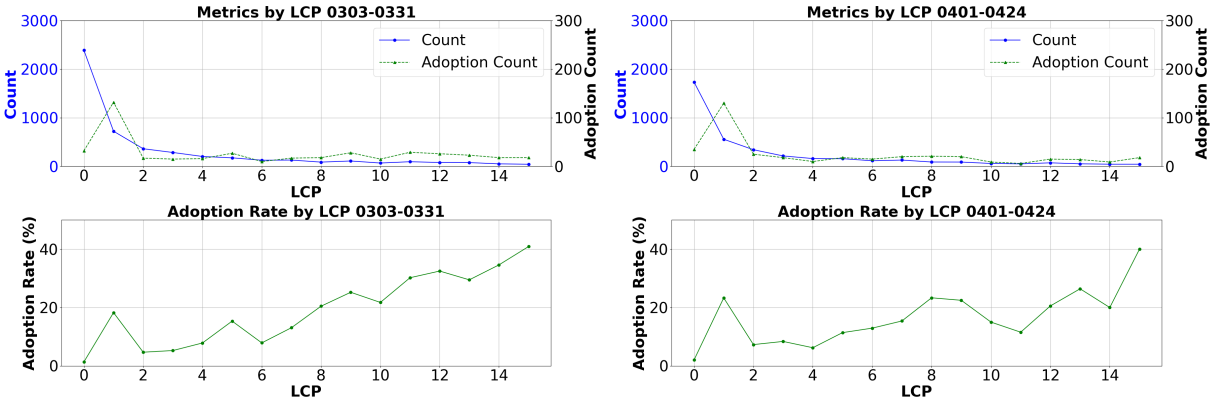

Figure 6: LCP Distribution and Its Relationship with Adoption Count and Adoption Rate

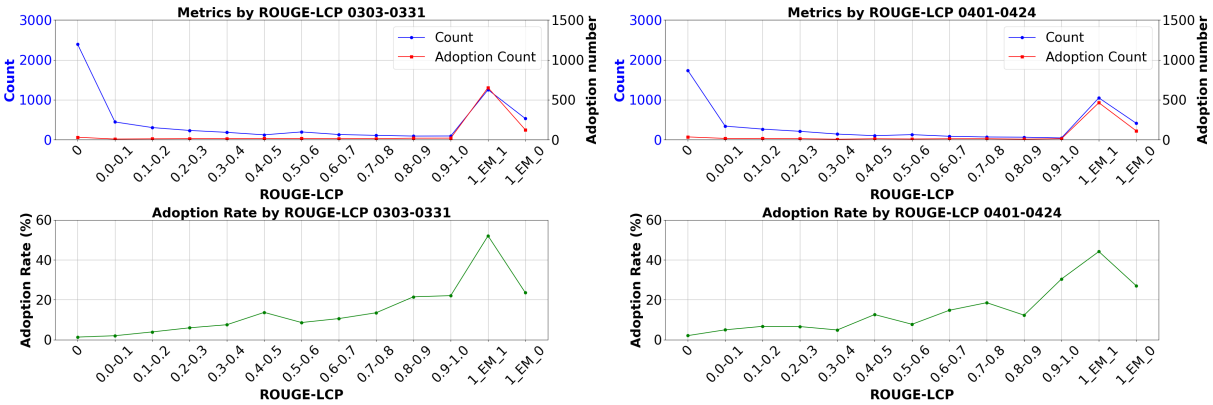

Figure 7: ROUGE-LCP Distribution and Its Relationship with Adoption Count and Adoption Rate

By observing the correlation heatmaps, we find that compared to commonly used code completion metrics, LCP shows the strongest correlation with the adoption rate across all time periods, with r values generally exceeding 0.7 and p-values all below 0.05; ROUGE-LCP ranks second. These results further validate that the proposed LCP and ROUGE-LCP metrics outperform general metrics in capturing user intent and adoption behavior.

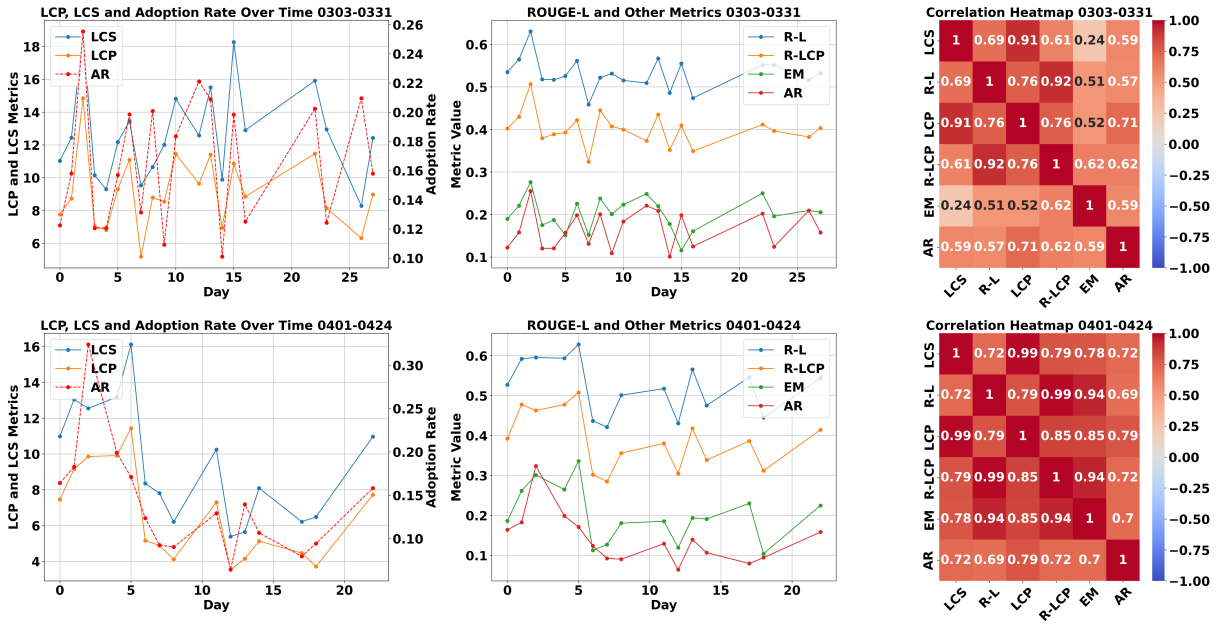

Figure 8: Daily Metric and Adoption Rate Distributions, Heatmap of Correlation Between Evaluation Metrics and Adoption Rate. R-L refers to ROUGE-L, R-LCP refers to ROUGE-LCP, and AR refers to Adoption Rate.