FastMCC: Multiple-hypotheses Code Completion with Natural Language Phrases as Alternative Suggestions

Zhensu Sun
Singapore Management University
Singapore
zssun@smu.edu.sg

Abstract

LLM-based code completion systems typically offer only one suggestion, often failing to cover developers' intent, especially with ambiguous context. Current multiple-hypotheses code completion, i.e., generating multiple full-code candidates to increase the possibility to hit the user intent, can mitigate this but with a price of high latency and developers' cognitive burden. To fill this gap, we propose FastMCC, which presents multiple hypotheses as concise natural language phrases instead of full code. It uses a two-stage inference: first generating an initial code suggestion followed by alternative intent phrases; then generating intent-aligned code if a phrase is selected. We design a lightweight fine-tuning framework to equip base code LLMs with this capability without compromising their original code completion performance. Experiments on three code LLMs show FastMCC maintains the pass rate of initial suggestions, while saving 76.6% generated tokens for multiple-hypotheses options.

ACM Reference Format:
Zhensu Sun. 2026. FastMCC: Multiple-hypotheses Code Completion with Natural Language Phrases as Alternative Suggestions. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-Companion '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3774748.3787758

1 Introduction

LLM-based code completion systems typically generate only one suggestion per request, but partial code often has multiple valid, functionally distinct completions, especially when intent is underspecified in the context[10]. For example, JavaScript JSON parsing can use native APIs, but developers may prefer third-party libraries due to efficiency concerns. Suggestions failing to match developer intent will lead to tedious manual revision cycles. A solution would be to have the model generate multiple candidates at once to cover potential intents as much as possible, i.e., *multi-hypotheses code completion*. This approach, however, is not popular in practice due to its limitation in latency (generating all full-length code candidates is slow and computationally expensive) and cognitive burden (developers have to review all code candidates).

To fill this gap, we propose FastMCC (Fast Multi-hypothesis Code Completion), a "natural language-as-alternative" paradigm. Instead of directly generating multiple full code blocks, it first presents concise phrase descriptions of alternative completions (e.g., "parse JSON with native API") and generates corresponding code only when a phrase is selected. This design addresses these limitations, as generating and understanding short phrases is fast, economical, and intuitive. To realize this, we design a lightweight training framework enabling a base code LLM to support this new paradigm without compromising its original capabilities. Correspondingly, we propose a two-stage inference method to efficiently achieve this paradigm in practical code completion scenarios.

For evaluation, we create a training dataset using GPT-5 and fine-tune three open-source code LLMs: Deepseek-Coder (6.7B), Qwen2.5-Coder (7B), and Seed-Coder (8B). We evaluate these models on HumanEval-infilling, focusing on **effectiveness**, **alignment**, and **efficiency**. Our results show FastMCC not only maintains the Pass@1 for initial suggestions but also generates, on average, **1.19 new valid solutions** (out of 5), a 5-time increase over the 0.24 from base models. A human study confirms a **64.1% match rate** for our second stage, validating that the generated code closely matches the chosen intent. The approach is highly efficient, saving **76.6%** of tokens compared to naively generating 5 full candidates.

2 Background and Related Work

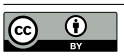
Multi-hypothesis code completion generates multiple candidates simultaneously, covering a wider range of hidden intents and increasing the likelihood of aligning with the developer's needs. For example, GitHub Copilot can show alternative suggestions in a new tab if the initial one is rejected [3]. This is typically implemented by sampling strategies such as beam search to produce diverse outputs.

In our research community, studies focuses on optimizing these generated code candidates through strategies like reranking [1, 8, 11], filtering [6], or advanced sampling [13]. While these methods improve the quality of the candidate list, they do not address the fundamental issues of computational cost and cognitive burden.

3 Method

FastMCC revolves around two core components: a lightweight fine-tuning framework that preserves base model capability while adding multi-hypothesis support and a two-stage inference mechanism balancing efficiency and intent coverage.

Fine-tuning Framework: The framework extends the widely-used Fill-in-the-Middle (FIM) paradigm. While traditional FIM data is (`Prefix`, `Suffix`, `Middle`), we add two elements, `Desc` and `Alters`. Specifically, *Desc* is a concise natural language phrase summarizing the Middle code's intent (e.g., *"parse JSON with native API"*). *Alters* is multiple natural language phrases (separated by a delimiter, e.g., line break), with each describing a different, but valid completion.



ICSE-Companion '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2296-7/2026/04
https://doi.org/10.1145/3774748.3787758

To train the model, we format these five elements paired with special tokens (denoted using square brackets) to define the input-output pair for a supervised fine-tuning task. Specifically, the input provided to the model consists of the code context (`Prefix`, `Suffix`) and the guiding intent (`Desc`):

```
[BOS] Prefix [PRE] Suffix [SUF] Desc [DES]
```

The model learns to generate the output (the `Middle` matching the `Desc`, followed by `Alters`):

```
Middle [MID] Alters [EOS]
```

This trains the model to generate code conditioned on natural language, which is critical for the two-phase inference.

Two-phase Inference: FastMCC's inference operates in two phases. Phase 1 generates a primary completion and alternative intents. Phase 2 (optional) is triggered if the user selects an intent, generating code completion that matches it.

For Phase 1 (Primary Suggestion), given context $Prefix$ and $Suffix$, the input prompt uses an empty description:

```
Prefix [PRE] Suffix [SUF] [DES]
```

The output follows the training format. The sequence before `[MID]` is the primary completion. Latency to this point is nearly identical to a base model's single completion. The inference continues past `[MID]` to produce natural language descriptions, which are split and displayed as options for developers to choose.

Phase 2 (Alternative Suggestion) is activated when the user selects an alternative. A new prompt is then built:

```
Prefix [PRE] Suffix [SUF] Chosen Alter [DES]
```

This step is efficient, since it reuses the model's attention key-value cache of Prefix and Suffix from Phase 1. The model only needs to compute for the subsequent description tokens (i.e., the chosen `Alter` and `[DES]`). It then generates the intent-specific code, where the generation stops at `[MID]`.

4 Experiments and Results

We create a training dataset (122.9k samples) using GPT-5 based on snippets from OSS-Instruct [12] and evol-codealpaca [7] to train three LLMs: DeepSeek-Coder (6.7B) [4], Qwen-2.5-Coder (7B) [5], and Seed-Coder (8B) [9]. For each snippet, we randomly select 1-3 lines as the middle span and instruct GPT-5 to generate Desc and Alters. Models are evaluated on HumanEval-infilling [2] for effectiveness, alignment, and efficiency.

Effectiveness: We evaluate effectiveness assuming that a given context may have multiple valid solutions, and the system's goal is to generate more diverse, valid options to cover the user's unknown intent. **Table 1** shows FastMCC achieves this without degrading primary suggestion quality, as the 'Initial Pass@1' performance is maintained or even improved (e.g., DS-Coder-FastMCC 76.6% vs. base 70.2%). The core benefit is in generating diverse valid alternatives (i.e., passing test cases but differing from the initial suggestion). Our models find an average of 1.19 new valid solutions ('#Valid') among 5 alternatives, approximately 5 times more than the 0.24 average from base models. This significantly boosts total coverage ('Pass@1+5'), pushing Qwen-Coder to 85.9% (from 80.4%) and DeepSeek-Coder to 83.3% (from 74.0%), demonstrating its ability to cover a wider range of potential user intents.

Table 1: Effectiveness comparison with 5 alternative suggestions on HumanEval-infilling

Model	Version	Initial	Alternatives (k=5)	
		Pass@1	Pass@1+5	#Valid
DS-Coder	base	70.2%	74.0%	0.22
	FastMCC	**76.6%**	**83.3%**	**1.26 (x5.7)**
Qwen	base	76.3%	80.4%	0.23
	FastMCC	**76.9%**	**85.9%**	**1.17 (x5.1)**
Seed	base	81.4%	86.5%	0.28
	FastMCC	**81.4%**	**89.1%**	**1.15 (x4.1)**

Alignment: We evaluated the alignment of Phase 2, i.e., whether the model generates code that matches the user-selected alternative intent. We employed both an LLM-as-judge and a human study to assess this. The LLM-as-judge, using GPT-5, determined a **66.5%** match rate between the generated code and the selected intent. In our human annotation, two experienced annotators are required to consider the alignment of the code's logic with the alternative intent, irrespective of the code's functional correctness. The evaluators achieved an inter-rater agreement of 0.75, and all disagreements were resolved through discussion to reach a final consensus. The annotation achieves an alignment rate of **64.1%**. This confirms that the model has learned to condition its generation on the `Desc` field. Mismatches often occurred when the context was too restrictive. For example, for a gap only required 'else:', there is no room to implement other intent while maintain contextually valid.

Efficiency: We analyze FastMCC's efficiency from two aspects. First, for the primary suggestion (Phase 1), the overhead is negligible. The input prompt requires only one additional token (`[DES]`), causing minimal latency for the primary `Middle` code. Second, for providing multi-hypothesis options, our method is significantly more efficient than the naive approach (i.e., generating 5 full code candidates). By generating one full completion plus 5 concise natural language phrases (instead of 5 full completions), FastMCC saves an average of **76.6%** of the token cost across our evaluation. This makes it a highly economical and responsive solution for real-time IDE environments.

5 Conclusion and Future Work

In this paper, we address the dilemma between single-intent completions and impractical multi-code alternatives through **FastMCC**, a "natural language-as-alternative" paradigm balancing intent coverage, cognitive load, and computational efficiency. Our lightweight fine-tuning and two-stage inference allow a base code model to provide a primary completion plus concise alternatives, then generate aligned code if an alternative is selected. Experiments show FastMCC models maintain or improve primary Pass@1 while generating 5 times more new valid solutions than base models. For future work, we will further explore how to obtain high-quality data to achieve better method performance.

References

[1] Daniele Cipollone, Egor Bogomolov, Arie van Deursen, and Maliheh Izadi. 2025. TreeRanker: Fast and Model-agnostic Ranking System for Code Suggestions in IDEs. *arXiv preprint arXiv:2508.02455* (2025).

[2] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[3] GitHub Inc. 2025. *Getting Code Suggestions in Your IDE with GitHub Copilot.* https://docs.github.com/en/copilot/how-tos/get-code-suggestions/get-ide-code-suggestions#showing-multiple-suggestions-in-a-new-tab-1

[4] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[5] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-Coder Technical Report. arXiv:2409.12186 [cs.CL] https://arxiv.org/abs/2409.12186

[6] Jia Li, Yuqi Zhu, Yongmin Li, Ge Li, and Zhi Jin. 2024. Showing llm-generated code selectively based on confidence of llms. *arXiv preprint arXiv:2410.03234* (2024).

[7] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568* (2023).

[8] Zhicun Lyu, Xinye Li, Zheng Xie, and Ming Li. 2025. Top Pass: improve code generation by pass@ k-maximized code ranking. *Frontiers of Computer Science* 19, 8 (2025), 198341.

[9] ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, et al. 2025. Seed-coder: Let the code model curate data for itself. *arXiv preprint arXiv:2506.03524* (2025).

[10] Zhensu Sun, Xiaoning Du, Fu Song, Shangwen Wang, Mingze Ni, Li Li, and David Lo. 2025. Don't complete it! Preventing unhelpful code completion for productive and sustainable neural code completion systems. *ACM Transactions on Software Engineering and Methodology* 34, 1 (2025), 1–22.

[11] Zhihong Sun, Yao Wan, Jia Li, Hongyu Zhang, Zhi Jin, Ge Li, and Chen Lyu. 2024. Sifting through the chaff: On utilizing execution feedback for ranking the generated code candidates. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 229–241.

[12] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Empowering code generation with oss-instruct. *arXiv preprint arXiv:2312.02120* (2023).

[13] Yuqi Zhu, Jia Li, Ge Li, YunFei Zhao, Zhi Jin, and Hong Mei. 2024. Hot or cold? adaptive temperature sampling for code generation with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 437–445.