International Journal on Advanced Science Engineering Information Technology

Vol.15 (2025) No. 3
ISSN: 2088-5334

Enhancing OpenAPI Code Completion with Adaptive and Hierarchical Fill-in-the-Middle Transformations

Muhammad Irtaza Malik [a], Sookyun Kim [a], Jaechoon Jo [b,1], YeoChan Yoon [c,2]

[a] *Department of Computer Engineering, Jeju National University, Jeju, Republic of Korea*
[b] *Department of Computer Education, Jeju National University, Jeju, Republic of Korea*
[c] *Department of Artificial Intelligence, Jeju National University, Jeju, Republic of Korea*
Corresponding author: [1]jjo@jejunu.ac.kr; [2]ycyoon@jejunu.ac.kr

***Abstract*—The development and maintenance of RESTful APIs have become increasingly critical with the widespread adoption of the OpenAPI Specification, a de facto standard for API description. However, manual authoring of these API definitions remains laborious and prone to human error. The primary objective of this study is to enhance the efficiency and accuracy of API code completion through an automated framework that leverages advanced transformation techniques. Our investigation employs a newly proposed semantics-aware benchmark that offers a robust dataset for evaluating code completion performance. Our method integrates adaptive transformation rates, positional boundary biases, multi-segment transformations, and hierarchical permutations to address the intrinsic challenges of API specification generation. Specifically, adaptive transformation rates enable dynamic adjustments during code generation, while positional boundary biases and multi-segment transformations improve context preservation and structural coherence. Hierarchical permutations further facilitate the accurate mapping of complex API constructs. Experimental results demonstrate significant improvements in both accuracy and efficiency compared to traditional manual and automated methods. These findings indicate that the proposed approach not only reduces development time but also minimizes specification errors. Implications for further research include exploring the integration of our method with existing development environments and extending its applicability to a broader range of code generation tasks, ultimately contributing to more reliable and maintainable API development practices. The proposed framework sets a promising direction for future research in automated API specification generation.**

***Keywords*—OpenAPI; RESTful APIs; code completion; adaptive transformation; semantics-aware benchmark.**

Manuscript received 12 Oct. 2024; revised 29 Dec. 2024; accepted 19 Feb. 2025. Date of publication 30 Jun. 2025.

I. INTRODUCTION

In today's interconnected digital landscape, Application Programming Interfaces (APIs) play a pivotal role in enabling seamless interoperability among diverse software systems [1]. The widespread adoption of microservices architectures and web services has underscored the importance of standardized API protocols to support scalable and modular development [2], [3]. As organizations increasingly rely on APIs to integrate complex systems, well-designed and robust APIs are essential for fostering innovation and operational efficiency [4]. The shift toward digital transformation further amplifies the need for effective communication between software components, making API documentation a cornerstone of modern software engineering [5].

The OpenAPI Specification has become the de facto standard for documenting RESTful APIs, offering a structured framework to define endpoints, parameters, and responses [6]. Despite its advantages, such as enhanced interoperability and simplified API design, manually crafting OpenAPI definitions is labor-intensive and error-prone due to their nested and verbose nature [7], [8]. This challenge has driven the development of intelligent tools to automate the creation and maintenance of API documentation [9]. Modern Integrated Development Environments (IDEs) incorporate code completion features that boost developer productivity by generating context-aware code snippets [10]. Fill-in-the-Middle (FIM) techniques, which infer missing code segments based on surrounding context, have shown promise [11]. However, applying FIM to OpenAPI specifications reveals limitations in handling their hierarchical and multi-level structures [12], [13].

Recent advancements in Large Language Models (LLMs) have opened up new avenues for automated code generation, with potential applications in specialized domains, such as OpenAPI [14]. Yet, traditional FIM approaches often fail to address the semantic and structural complexities of API

documentation, leading to incomplete or inaccurate completions [15]. This paper proposes a novel framework that overcomes these shortcomings by introducing Adaptive Transformation Rates to balance segment complexity, Positional Boundary Biases to ensure semantically meaningful splits, Multi-Segment Transformations to process nested structures, and Hierarchical Permutations to capture relational dependencies. Experimental results on a semantics-aware OpenAPI benchmark demonstrate significant improvements over conventional methods, advancing the state of the art in API code generation.

II. Materials and Method

A. Literature Review

This review synthesizes foundational and recent advancements in code generation, language modeling, and API documentation, with a focus on addressing the challenges of OpenAPI code completion. The discussion covers transformer architecture, code infilling techniques, prompt engineering, domain adaptation, and evaluation metrics, providing a comprehensive backdrop for the proposed framework as shown in Figure 1.

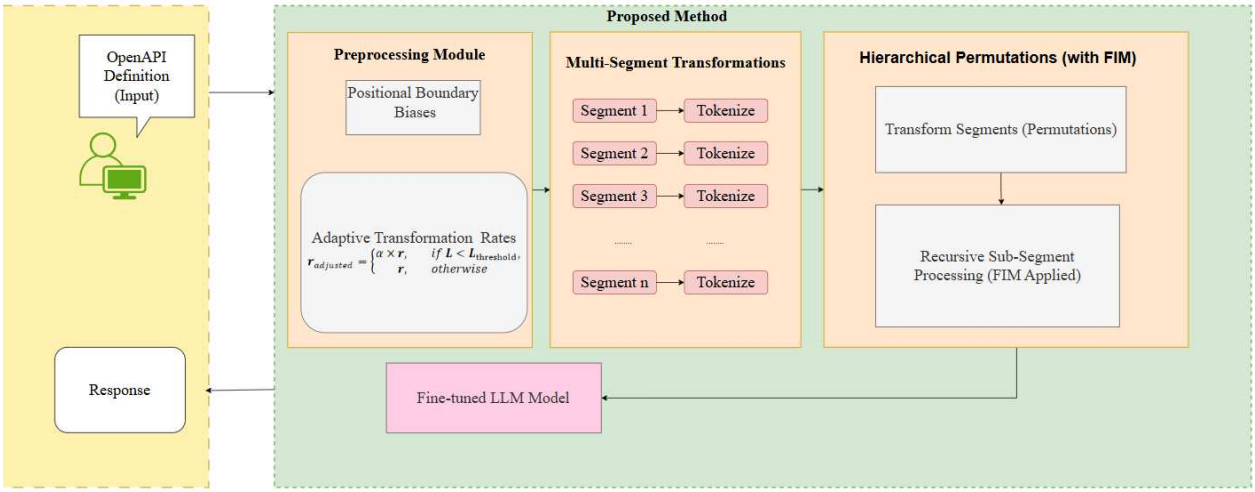

Fig 1 This diagram shows the working architecture of our system

1) Foundations of Code Generation and Language Models: The transformer architecture has revolutionized both natural language processing and code generation [16]. Bidirectional models like BERT [17] and RoBERTa [18] HEPH demonstrated the power of pre-training on large corpora. Sequence-to-sequence models, such as T5 [19] and BART [20], extended these capabilities to generative tasks, while autoregressive models like GPT-3 [21] showcased few-shot learning potential. Code-specific models, such as CodeGen [22], have advanced domain-specific pre-training, enabling robust code generation for programming tasks [23]. These developments provide the foundation for applying LLMs to API-related code generation [24].

2) Code Infilling Techniques: Traditional code generation relies on left-to-right decoding, which overlooks bidirectional context critical for infilling tasks [11]. Fill-in-the-Middle (FIM) techniques address this by conditioning on both preceding and following tokens, as demonstrated by InCoder [25]. However, FIM methods often assume uniform segment treatment, which is suboptimal for OpenAPI's nested structures [12]. Recent studies propose structure-aware infilling to handle hierarchical data, but challenges remain in addressing semantic boundaries [26], [27].

3) Prompt Engineering and Fine-Tuning Strategies: Prompt engineering significantly enhances LLM performance in code generation. Few-shot prompting [21] and chain-of-thought prompting [28] improve reasoning and context utilization. Domain-specific prompts further refining outputs for specialized tasks [29]. Fine-tuning strategies, such as parameter-efficient methods like AdapterFusion [30], BitFit [31], and LoRA [32], enable adaptation to niche domains like OpenAPI without extensive retraining. These approaches are critical for optimizing models for API documentation tasks [33].

4) Evaluation Metrics and Domain-Specific Benchmarks: Evaluating code generation requires metrics beyond traditional NLP standards like BLEU or ROUGE. Benchmarks like HumanEval and MultiPL-E focus on functional correctness across programming languages [34], [35]. For OpenAPI, semantic-aware evaluation is essential, with tools like diff detecting meaningful changes in API definitions [6]. Datasets such as The Stack [36] provide real-world samples for training and benchmarking domain-specific solutions, enabling robust validation of code completion frameworks [37].

5) Domain Adaptive Code Completion and Related Approaches: General-purpose tools like GitHub Copilot and AWS CodeWhisperer excel in mainstream programming but struggle with OpenAPI's structural complexity [2], [38]. Domain-adaptive approaches, such as those by [39], leverage decoupled databases to enhance specificity. A semantics-aware OpenAPI benchmark by [15] revealed that fine-tuned models, such as Code Llama, outperform commercial solutions, emphasizing the need for specialized methods [40]. Hierarchical transformers, as explored by [41], offer promise for multilevel API completion.

6) Proposed Enhancements: Building on adaptive computation [42] and hierarchical modeling [43], the proposed framework integrates Adaptive Transformation Rates, Positional Boundary Biases, Multi-Segment Transformations, and Hierarchical Permutations. These address the nested

relationships in OpenAPI definitions, drawing inspiration from sparsely-gated mixture-of-experts models [44] and hierarchical permutation learning [45]. This cohesive approach overcomes the limitations of legacy FIM methods, offering a robust solution for OpenAPI code completion.

B. Method

To resolve the shortcomings of legacy FIM techniques in generating accurate and structurally coherent OpenAPI definitions, we propose a multifaceted approach that directly addresses the hierarchy, complexity, and verbosity inherent to API specifications. By integrating context-aware transformations, targeted boundary identification, and dynamic adaptation to content size, our methods collectively deliver more reliable and semantically aligned OpenAPI completions. Below, we detail each core component of our framework.

1) Adaptive Transformation Rates:

As discussed in the Introduction, existing Fill-in-the Middle (FIM) solutions often underperform when faced with the hierarchical complexity and verbosity of OpenAPI definitions [6], [9]. While basic FIM strategies can work well for linear or uniformly structured code, the nested segments and variable sizes found in OpenAPI documents demand a more context-aware approach [12].

Conventional FIM approaches, as highlighted in the Related Works, typically employ a uniform transformation rate across all segments, leading to two key challenges. Short segments, such as simple parameter definitions, risk being over-transformed, which may introduce unnecessary changes or errors that distort the intended meaning, while longer and more complex segments—like endpoints with nested objects and parameters—might be under-transformed, resulting in omissions that fail to capture the document's intricate relationships and dependencies. To address these issues, we introduce a dynamic formula that defines a base transformation rate, r, which specifies the fraction of a segment subject to transformation; moreover, when a segment's length L falls below a user-defined threshold Lthreshold, a scaling factor α(<1) is applied to reduce r accordingly, ensuring that each segment is transformed proportionally to its complexity and significance.

$$\boldsymbol{r}_{adjusted} = \begin{cases} \alpha \times \boldsymbol{r}, & if\ \boldsymbol{L} < \boldsymbol{L}_{\text{threshold}}, \\ \boldsymbol{r}, & otherwise \end{cases} \quad (1)$$

Here, $\boldsymbol{r}$ is the default degree of transformation, and α ensures smaller segments remain relatively intact. For larger, more complex sections, $\boldsymbol{r}_{\text{adjusted}}$ remains at $\boldsymbol{r}$, guaranteeing that crucial nested relationships are adequately restructured.

Practical benefits include balanced editing, where smaller segments receive milder edits that avoid the over-transformation pitfalls identified in the Introduction; reduced errors, since large, nested blocks common in OpenAPI definitions retain higher transformation rates to preserve their hierarchical integrity; and context sensitivity, as the approach aligns with domain-adaptive methods highlighted in related works [12], ensuring that each segment's transformation is proportional to its complexity and significance within the API specification. In essence, Adaptive Transformation Rates provide a measured, context-sensitive editing mechanism that addresses the FIM limitations detailed in both the Introduction and Related Works, and by dynamically calibrating transformation rates, our solution mitigates over and under-transformation issues, paving the way for more reliable OpenAPI code completions.

2) Positional Boundary Biases:

Fill-in-the-Middle (FIM) techniques often split code based on arbitrary lines or tokens, overlooking semantically meaningful regions. In OpenAPI definitions, these regions include paths, components, and parameter blocks—all of which are critical to preserving coherent structure.

Algorithm 1 Positional Boundary Selection

Require: Content C, number of boundaries n

- Identify semantic boundaries $B_{semantic}$ in C
- Assign higher selection probabilities to $B_{semantic}$
- Select n boundaries based on the biased probabilities

The process begins by identifying semantic boundaries, which involves locating crucial markers such as paths: and components: that clearly delineate major API elements. These markers are then assigned higher probabilities, ensuring that they are favored over purely syntactic boundaries when making selections. With these enhanced probabilities, a predetermined number of splits are randomly chosen, but each split is weighted according to the likelihood of being a semantic boundary. This method ensures that the splits align closely with the inherent domain-specific structure of OpenAPI, thereby reducing the fragmentation of important elements. By emphasizing semantic boundaries, the approach not only preserves the integrity of the API but also enhances the relevance and accuracy of any subsequent transformations.

3) Multi-Segment Transformations:

Traditional Fill-in-the-Middle methods often rely on a three-segment (prefix, middle, suffix) model. However, OpenAPI definitions can contain multiple semantic blocks such as endpoints, components, and parameter set that exceed this simplistic partitioning.

Algorithm 2 Multi-Segment Transformation

Require: Content C, boundaries B, tokenizer T

- Split C into segments $S = \{s_1, s_2, \ldots, sk\}$
- for each segment s_i in S do
- Tokenize s_i using T
- end for
- Permute segments based on a transformation strategy

Content C is divided into multiple segments S using clearly identified boundaries, ensuring that each portion is treated appropriately—whether it is a simple parameter or an extensive path—before moving on to further processing. Each segment is then tokenized to enable granular editing, and a carefully chosen transformation strategy is applied to permute these segments, thereby exposing the model to varied yet structurally coherent inputs.

This approach offers significant benefits, including finer granularity that aligns well with the nested nature of OpenAPI specifications and the preservation of context by addressing each segment's unique requirements without falling into the

pitfalls of over or under-segmentation. Moreover, this flexible method facilitates subsequent steps, such as Hierarchical Permutations, by enabling operations to be conducted on logically discrete pieces of the overall specification, ultimately enhancing the robustness and accuracy of the transformation process.

4) Hierarchical Permutations:

OpenAPI specifications frequently contain deeply nested structures, such as embedded schemas, parameters, and sub-components, that are arranged in intricate hierarchical relationships. This complexity demands a transformation approach capable of addressing each level of the specification with both precision and sensitivity to context. While multi-segment transformations enhance granularity by breaking the document into manageable pieces for individual editing, Hierarchical Permutations extend this idea further by recursively applying transformations to sub-segments. This recursive process ensures that not only the primary segments but also every nested level—ranging from overarching schemas to the smallest parameters—receives adjustments that are finely tuned to its specific context and structural role. By meticulously refining each layer of the hierarchy, this method preserves the integrity of the original document, maintains semantic relationships, and mitigates the risk of over- or under-transformation. Ultimately, this comprehensive approach yields more robust and reliable code completions, ensuring that even the most complex and nested components of an OpenAPI specification are accurately and effectively transformed.

Algorithm 3 Hierarchical Permutation

Require: Segments S, max depth D_{max}, current depth I

- if $d < D_{max}$ then
- for each segment s_i in S do
- Apply transformation to s_i iv.
- Recursively call Algorithm 3 on sub-segments of s_i with $d + 1$
- end for
- end if

Starting at the top-level segments, the method applies transformations—such as fill-in or token-level edits—to establish a foundation. For segments with deeper layers, like complex YAML objects or references, it recursively descends until the maximum allowed depth is reached. This recursive process ensures that each nested tier retains the structural context of its parent, thereby reducing the risk of flattening relationships or losing critical schema references. Moreover, because the segments have already been partitioned and possibly permuted, hierarchical recursion ensures that all relevant sub-segments are appropriately refined, thereby maintaining the semantic integrity of the entire OpenAPI definition. Overall, this layer-by-layer refinement process accommodates the multi-level constructs inherent in OpenAPI, producing code completions that remain coherent, accurate, and faithful to the nested structure of the specification while enhancing adaptability and precision in handling complex nested structures.

5) Datasets:

We employed two distinct datasets for our evaluation to rigorously test our model's performance across different aspects of code generation. The first dataset, OpenAPI Completion Refined, was collected from the APIs.guru directory and comprises 990 OpenAPI definitions spanning diverse domains and complexity levels. This dataset was specifically designed for fine-tuning Code Llama on the OpenAPI completion task, capturing the real-world intricacies and hierarchical structures unique to API specifications. Its diversity and detailed structure provide a robust foundation for evaluating domain-specific performance.

The second dataset, Human Eval Reference, is built upon the code generation tasks discussed in MultiPL-E and expands upon the well-known HumanEval benchmark. This dataset measures functional correctness by synthesizing programs from docstrings, effectively testing language comprehension, simple algorithms, and mathematical reasoning. It serves as a complementary, general code generation benchmark that allows us to assess broader adaptability and correctness beyond the OpenAPI domain. By combining OpenAPI Completion Refined for domain-specific performance with Human Eval Reference for general functional correctness, we ensure a balanced and comprehensive evaluation of our model's code generation capabilities, addressing both specialized and universal coding challenges.

6) Model Training:

We fine-tuned Code Llama 7B [36] and Code Llama 13B [37] using a batch size of 32, with gradient accumulation (4 steps) to achieve an effective batch size of 128. A cosine learning rate schedule peaked at 2×10^{-4}, with a 10% warm-up ratio. Optimization followed AdamW (β_1=0.9, β_2=0.999, weight decay =0.01= 0.01=0.01) over three epochs (or until convergence). We set the context length to 4096 tokens and applied mixed-precision (FP16) training on two NVIDIA RTX A6000 GPUs. These parameters align with standard fine-tuning practices [2], ensuring consistency with prior baselines while facilitating our novel adaptive strategies (e.g., Adaptive Transformation Rates and Positional Boundary Biases) for OpenAPI code completion.

7) Evaluation Metrics:

We evaluate model performance on two key metrics: Correctness and Validity. Correctness measures the percentage of completions that are semantically identical to the ground truth— factoring in structural equivalence and ignoring minor stylistic differences (e.g., key ordering or optional quotes).

Validity gauges the percentage of completions yielding syntactically valid OpenAPI definitions, ensuring each suggestion adheres to the specification's formal requirements. Taken together, these metrics provide a balanced view of both functional accuracy and structural compliance in generated OpenAPI completions.

III. Results and Discussion

This section presents a detailed comparison of our fill-in-the-middle (FIM) enhancements against baseline FIM approaches, highlighting improvements in correctness, validity, and training efficiency across varying dataset complexities and error profiles. All experiments focus on

Llama7B and Llama13B, leveraging our Adaptive Transformation Rates, Positional Boundary Biases, Multi-Segment Transformations, and Hierarchical Permutations for OpenAPI code generation.

A. Results

The following is the observed behavior of our system across different evaluation matrices.

1) Performance vs. Dataset Complexity:

Figure 2 plots correctness percentages for Baseline FIM Llama7B (blue) and Our FIM Llama7B (green) across four complexity bins (C1: Low, C2: Moderate, C3: High, C4: Extreme). Although both models experience a natural decline in correctness as definitions become more complex (e.g., deeper nesting, extensive references), our method consistently outperforms the baseline in every bin. Notably, we retain a higher correctness margin even under Extreme complexity (C4), indicating that Adaptive Transformation Rates and Hierarchical Permutations effectively maintain structural fidelity in challenging OpenAPI scenarios.

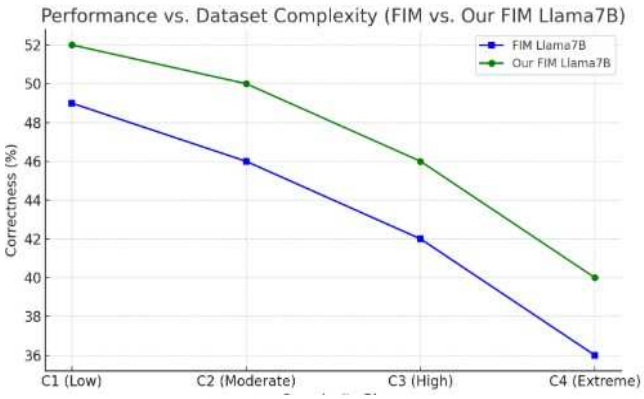

Fig. 2 Line chart comparing the performance of FIM

2) Error/Failure Types Comparison:

Figure 3 compares error counts for three major failure modes syntax errors (blue), broken references (orange), and missing fields (green)—between Baseline FIM Llama7B and Our FIM Llama7B, with the stacked bars revealing several key trends that highlight the advantages of our approach. Our model not only exhibits a lower overall error volume, but it also shows a significant reduction in errors within each category, particularly in syntax errors and missing fields. This performance improvement suggests that the implementation of Positional Boundary Biases and Multi-Segment Transformations substantially enhances the model's capacity to preserve crucial references and manage nested elements more effectively.

By accurately maintaining the hierarchical structure and semantic integrity of complex API specifications, our approach minimizes common pitfalls associated with OpenAPI generation. These improvements indicate that the enhanced model is better equipped to handle both micro-level editing issues and macro-level structural complexities, resulting in more reliable and robust OpenAPI code completions that can ultimately simplify debugging and maintenance in practical, real-world applications.

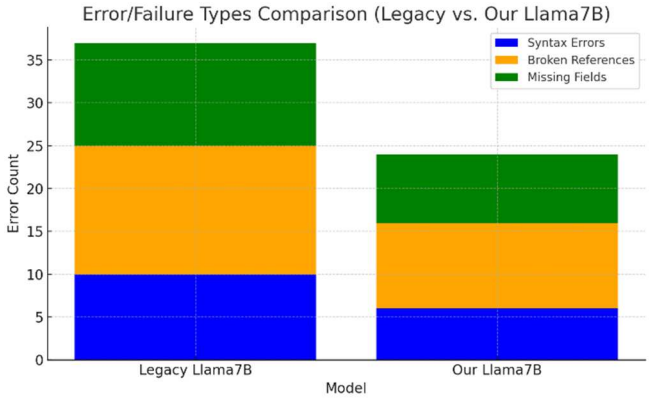

Fig. 3 Bar chart comparing error/failure types for FIM

3) Training Steps vs. Accuracy:

Figure 4 provides a side-by-side view of correctness (blue) and validity (green) for four distinct configurations: Baseline FIM Llama7B, Baseline FIM Llama13B, Our FIM Llama7B, and Our FIM Llama13B. The data confirm that both correctness and validity improve with our approach relative to the baseline FIM methods, with the Llama7B variant showing particularly large gains—reaching up to 48% correctness and 87% validity in some subsets.

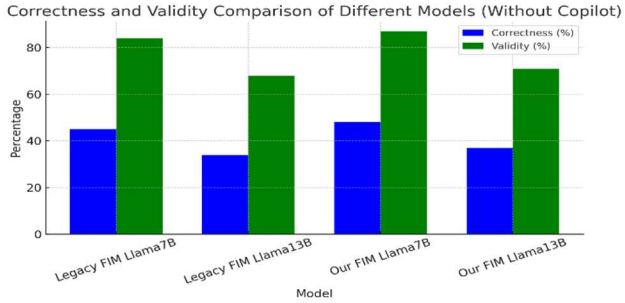

Fig. 4 Bar chart comparing correctness and validity for different models

This finding aligns with observations in prior studies [12], suggesting that models with smaller parameter counts can benefit substantially from carefully tuned, domain-specific enhancements. Such targeted modifications not only improve performance metrics but also underscore the potential of optimized configurations to achieve higher reliability and efficiency without the need for larger, more complex models.

4) Training Steps vs. Accuracy:

Figure 5 illustrates the training efficiency by plotting accuracy against training steps for both Baseline FIM Llama7B and Our FIM Llama7B. As the number of training steps increases, both models show a clear performance improvement, yet our approach consistently maintains a higher accuracy at every checkpoint from 10k to 50k steps. This steady lead suggests that our method is more effective in capturing OpenAPI-specific patterns, resulting in a more robust learning process.

By the final epoch, Our FIM Llama7B converges at an accuracy of 47.5% compared to the baseline's 45%, highlighting not only the accelerated convergence but also the enhanced reliability of our training strategy. These results underscore the significant benefits of our approach in efficiently and accurately understanding the complexities inherent in the data.

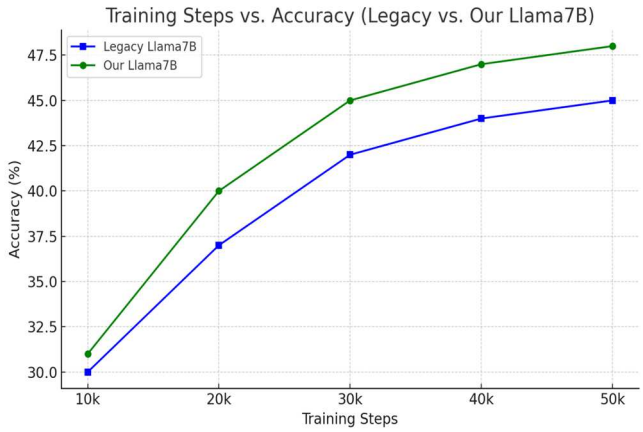

Fig. 5 The line plot of training steps vs. accuracy for FIM Llama7B and Our Llama7B

B. *Summary of Results*

Overall, these findings validate our novel fill-in-the-middle enhancements by demonstrating higher correctness across various dataset complexities, as shown in Figure 2, and by achieving lower error rates in syntax, references, and missing fields according to Figure 3. In addition, our approach yields improved validity and consistency for both 7B and 13B Llama models, as indicated in Figure 4, and it facilitates faster and higher convergence during training, as demonstrated in Figure 5. By incorporating Adaptive Transformation Rates, Positional Boundary Biases, Multi-Segment Transformations, and Hierarchical Permutations, we effectively address the inherent hierarchical complexity of OpenAPI definitions, resulting in consistent performance gains over baseline FIM solutions, particularly for smaller model architectures. These comprehensive improvements not only enhance accuracy and reliability across diverse conditions but also pave the way for more robust and scalable API specification processing.

C. *Discussion*

Table 1 illustrates the correctness and validity metrics for GitHub Copilot, two Baseline FIM Llama models,

TABLE I
PERFORMANCE COMPARISON BETWEEN MODELS

Model	Correctness	Validity	Correctness Avg	Validity Avg
GitHub Copilot	29.0	68.0	29.0	68.0
Legacy FIM Llama7B Model	45.0	84.0	32.0	63.1
Legacy FIM Llama13B Model	34.0	68.0	30.2	64.0
Our FIM Llama7B Model	48.0	87.0	33.4	64.9
Our FIM Llama13B Model	37.0	71.0	31.5	66.0

Our FIM-enhanced Llama models reveal a clear trend in performance differences that underscores the utility of fill-in-the-middle strategies for OpenAPI. Although GitHub Copilot attains 29.0% correctness and 68.0% validity, the Baseline FIM Llama7B model surpasses these figures with 45.0% correctness and 84.0% validity, while our FIM Llama7B further improves correctness to 48.0% and validity to 87.0%, representing a notable 19% improvement in correctness over Copilot and a 3% enhancement over the baseline model. This trend, as also depicted in the line plot of training steps versus accuracy in Figure 5, persists when comparing the Baseline FIM Llama13B to our FIM Llama13B, although the gains are more modest with the larger model. Such differences suggest that our adaptive fill-in-the-middle approach—which incorporates adaptive transformation rates, positional boundary biases, multi-segment transformations, and hierarchical permutations—provides disproportionately larger benefits in smaller model configurations, as evidenced by the error analysis in Figure 3 and complexity-based performance in Figure 2. These evaluations demonstrate the potential of our method to yield completions that are both structurally valid and semantically accurate, even under the challenging constraints of OpenAPI definitions, and they further corroborate our observations from training efficiency and error-type comparisons by underscoring the consistent advantage of our FIM enhancements over both a commercial tool like GitHub Copilot and baseline FIM solutions.

IV. CONCLUSION

In this work, we presented a novel approach to OpenAPI code completion that transcends traditional fill-in-the-middle techniques. By incorporating Adaptive Transformation Rates, Positional Boundary Biases, Multi-Segment Transformations, and Hierarchical Permutations, our method effectively addresses the challenges posed by the nested and complex nature of OpenAPI specifications. Our approach dynamically adjusts transformations based on content length and complexity, ensures semantic coherence by selecting meaningful boundaries, and leverages multi-level segmentation to preserve hierarchical relationships.

Experimental results demonstrate that our enhanced FIM Llama models consistently outperform both baseline FIM approaches and commercial tools such as GitHub Copilot in terms of correctness and validity. Notably, the improvements are most pronounced in the 7B model, indicating that our technique can optimize smaller architectures and reduce computational overhead while maintaining robust performance. This underscores the potential of our method to offer both efficiency and accuracy in practical API development scenarios.

Overall, our study not only contributes a robust framework for OpenAPI code completion but also lays the groundwork for further research into domain-specific adaptations of large language models. Future work will focus on refining these adaptive strategies and exploring their application to other complex, structured domains.

ACKNOWLEDGMENT

This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Education (No. RS-2023-00245316).

REFERENCES

[1] F. Di Lauro, S. Serbout, and C. Pautasso, "Towards large-scale empirical assessment of web APIs evolution," in *Proc. Int. Conf. Web Eng. (ICWE)*, M. Brambilla, R. Chbeir, F. Frasincar, and I. Manolescu, Eds. Cham, Switzerland: Springer, 2021, pp. 128-143, doi:10.1007/978-3-030-74296-6_10.

[2] M. Kim, T. Stennett, D. Shah, S. Sinha, and A. Orso, "Leveraging large language models to improve REST API testing," in *Proc. 44th Int. Conf. Softw. Eng.: New Ideas Emerging Results (ICSE-NIER)*, 2024, pp. 37–41, doi: 10.1145/3639476.3639769.

[3] T. Espinha, A. Zaidman, and H.-G. Gross, "Web API growing pains: Loosely coupled yet strongly tied," *J. Syst. Softw.*, vol. 100, pp. 27-43, Feb. 2015, doi: 10.1016/j.jss.2014.10.014.

[4] J. M. Rojas and G. Fraser, "Code defenders: A mutation testing game," in *Proc. 9th Int. Conf. Softw. Testing, Verification Validation Workshops (ICSTW)*, Apr. 2016, doi: 10.1109/icstw.2016.43.

[5] A. Neumann, N. Laranjeiro, and J. Bernardino, "An analysis of public REST web service APIs," *IEEE Trans. Serv. Comput.*, vol. 14, no. 4, pp. 957-970, Jul. 2021, doi: 10.1109/TSC.2018.2847344.

[6] OpenAPI Initiative, "OpenAPI Specification," 2023. [Online]. Available: https://swagger.io/specification/.

[7] S. Gao, X. Jiang, Q. Wu, X. Wang, C. Lyu, and L. Lyu, "GT-SimNet: Improving code automatic summarization via multi-modal similarity networks," *J. Syst. Softw.*, vol. 194, Dec. 2022, doi:10.1016/j.jss.2022.111495.

[8] T. Zhu, Z. Li, M. Pan, C. Shi, T. Zhang, Y. Pei, and X. Li, "Revisiting information retrieval and deep learning approaches for code summarization," in *Proc. 45th Int. Conf. Softw. Eng. Companion (ICSE-Companion)*, Melbourne, Australia, 2023, pp. 328-329, doi:10.1109/ICSE-Companion58688.2023.00091.

[9] Z. Zhou, H. Yu, G. Fan, Z. Huang, and K. Yang, "Summarizing source code with hierarchical code representation," *Inf. Softw. Technol.*, vol. 143, Mar. 2022, doi: 10.1016/j.infsof.2021.106761.

[10] Z. Tang, X. Shen, C. Li, J. Ge, L. Huang, Z. Zhu, and B. Luo, "AST-Trans: Code summarization with efficient tree-structured attention," in *Proc. 44th Int. Conf. Softw. Eng. (ICSE)*, Pittsburgh, PA, USA, 2022, pp. 150-162, doi: 10.1145/3510003.3510224.

[11] D. Fried et al., "InCoder: A generative model for code infilling and synthesis," 2022, *arXiv:2204.05999*. [Online]. Available: https://arxiv.org/abs/2204.05999

[12] Y. Gao, H. Zhang, and C. Lyu, "Encosum: Enhanced semantic features for multi-scale multi-modal source code summarization," *Empir. Softw. Eng.*, vol. 28, no. 5, Sep. 2023, doi: 10.1007/s10664-023-10384-x.

[13] M. Geng et al., "Interpretation-based code summarization," in *Proc. 31st IEEE/ACM Int. Conf. Program Comprehension (ICPC)*, May 2023, pp. 113-124, doi: 10.1109/ICPC58990.2023.00026.

[14] B. Petryshyn and M. Lukoševičius, "Optimizing large language models for OpenAPI code completion," 2024, *arXiv:2405.15729*. [Online]. Available: https://arxiv.org/abs/2405.15729

[15] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics (ACL)*, Online, 2020, pp. 4998-5007, doi: 10.18653/v1/2020.acl-main.449.

[16] A. Vaswani et al., "Attention is all you need," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst. (NIPS)*, Long Beach, CA, USA, 2017, pp. 6000-6010.

[17] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pretraining of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Human Lang. Technol. (NAACL-HLT)*, Minneapolis, MN, USA, 2019, pp. 4171-4186.

[18] Y. Liu et al., "RoBERTa: A robustly optimized BERT pretraining approach," 2019, *arXiv:1907.11692*. [Online]. Available: https://arxiv.org/abs/1907.11692

[19] C. Raffel, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 140, pp. 1–67, 2020.

[20] M. Lewis et al., "BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics (ACL)*, Online, 2020, pp. 7871-7880, doi: 10.18653/v1/2020.acl-main.703.

[21] T. B. Brown et al., "Language models are few-shot learners," in *Proc. 34th Int. Conf. Neural Inf. Process. Syst. (NeurIPS)*, Vancouver, BC, Canada, 2020, pp. 1877-1901.

[22] E. Nijkamp et al., "CodeGen: An open large language model for code with multi-turn program synthesis," 2022, *arXiv:2203.13474*. [Online]. Available: https://arxiv.org/abs/2203.13474.

[23] M. Chen et al., "Evaluating large language models trained on code," 2021, *arXiv:2107.03374*. [Online]. Available: https://arxiv.org/abs/2107.03374.

[24] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proc. 6th ACM SIGPLAN Int. Symp. Mach. Program. (MAPS@PLDI)*, San Diego, CA, USA, 2022, pp. 1-10, doi: 10.1145/3520312.3534862.

[25] C. Koutcheme, S. Sarsa, J. Leinonen, A. Hellas, and P. Denny, "Automated program repair using generative models for code infilling," in *Proc. 24th Int. Conf. Artif. Intell. Educ. (AIED)*, vol. 13916, Cham, Switzerland: Springer, 2023, pp. 902-914, doi:10.1007/978-3-031-36272-9_74.

[26] S. Lu et al., "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," 2021, *arXiv:2102.04664*. [Online]. Available: https://arxiv.org/abs/2102.04664.

[27] Z. Feng, D. Guo, D. Tang, and N. Duan, "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. Findings Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, Online, 2020, pp. 1536-1547, doi: 10.18653/v1/2020.findings-emnlp.139.

[28] J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," 2022, *arXiv:2201.11903*. [Online]. Available: https://arxiv.org/abs/2201.11903.

[29] Q. Luo et al., "RepoAgent: An LLM-powered open-source framework for repository-level code documentation generation," 2024, *arXiv:2402.16667*. [Online]. Available: https://arxiv.org/abs/2402.16667.

[30] J. Pfeiffer et al., "AdapterFusion: Non-destructive task composition for transfer learning," in *Proc. 16th Conf. Eur. Chapter Assoc. Comput. Linguistics (EACL)*, Online, 2021, pp. 487-503, doi:10.18653/v1/2021.eacl-main.39.

[31] E. B. Zaken, Y. Goldberg, and S. Ravfogel, "BitFit: Simple parameter-efficient fine-tuning for transformer-based masked language-models," in *Proc. 60th Annu. Meeting Assoc. Comput. Linguistics (ACL)*, Dublin, Ireland, 2022, pp. 1-9, doi: 10.18653/v1/2022.acl-short.1.

[32] E. J. Hu et al., "LoRA: Low-rank adaptation of large language models," 2021, *arXiv:2106.09685*. [Online]. Available: https://arxiv.org/abs/2106.09685.

[33] T. Liu, Y. Hu, W. Wu, Y. Wang, K. Xu, and Q. Yin, "DAP: Domain-aware prompt learning for vision-and-language navigation," 2023, *arXiv:2311.17812*. [Online]. Available: https://arxiv.org/abs/2311.17812.

[34] Z. Zhou, H. Yu, G. Fan, Z. Huang, and X. Yang, "Towards retrieval-based neural code summarization: A meta-learning approach," *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 3008-3031, Apr. 2023, doi:10.1109/TSE.2023.3238161.

[35] F. Cassano et al., "MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation," *IEEE Trans. Softw. Eng.*, vol. 49, no. 7, pp. 3675–3691, Jul. 2023, doi: 10.1109/TSE.2023.3267446.

[36] BigScience, "The Stack dataset," 2023. [Online]. Available: https://huggingface.co/datasets/bigcode/the-stack.

[37] Z. Tang et al., "Domain adaptive code completion via language models and decoupled domain databases," in Proc. 38th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE), Sep. 2023, pp. 421-433, doi:10.1109/ASE56229.2023.00076.

[38] Y. Li, D. Choi, J. Chung, and N. Kushman, "Competition-level code generation with AlphaCode," *Science*, vol. 378, no. 6624, pp. 1092-1097, Dec. 2022, doi: 10.1126/science.abq1158.

[39] J. Liu, L. Liu, J. Park, and W.-P. Chen, "Web API search: Discover web API and its endpoint with natural language queries," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, W.-S. Ku, Y. Kanemasa, M. A. Serhani, and L.-J. Zhang, Eds. Cham, Switzerland: Springer, 2020, pp. 96-113, doi: 10.1007/978-3-030-59618-7_7.

[40] F. D. Lauro, S. Serbout, and C. Pautasso, "A large-scale empirical assessment of web API size evolution," *J. Web Eng.*, Nov. 2022, doi:10.13052/jwe1540-9589.2167.

[41] S. Gao et al., "Code structure-guided transformer for source code summarization," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, pp. 1-32, Jan. 2023, doi: 10.1145/3522674.

[42] M. Dehghani et al., "Universal transformers," 2018, *arXiv:1807.03819*. [Online]. Available: https://arxiv.org/abs/1807.03819

[43] Z. Dai et al., "Transformer-XL: Attentive language models beyond a fixed-length context," in *Proc. 57th Annu. Meeting Assoc. Comput. Linguistics (ACL)*, Florence, Italy, 2019, pp. 2978-2988, doi:10.18653/v1/P19-1285.

[44] N. Shazeer et al., "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," 2017, *arXiv:1701.06538*. [Online]. Available: https://arxiv.org/abs/1701.06538.

[45] R. Umagami, Y. Ono, Y. Mukuta, and T. Harada, "HiPerformer: Hierarchically permutation-equivariant transformer for time series forecasting," 2023, *arXiv:2305.08073*. [Online]. Available: https://arxiv.org/abs/2305.08073