LLM-based API Argument Completion with Knowledge-Augmented Prompts

Anonymous Author(s)

ACM Reference Format:
Anonymous Author(s). 2025. LLM-based API Argument Completion with Knowledge-Augmented Prompts. In . ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 SUPPLEMENTARY RESULTS

1.1 Evaluation Process and Matrices

We evaluated `APICopilot` and baselines using *Precision@k* and *Recall@k*, standard metrics for API argument completion utilized by ARist. *Precision@k* is the ratio of correct top-k recommendations for supported argument types. *Recall@k* is the proportion of correctly identified arguments within the top-k suggestions relative to all arguments to be completed.:

$$\text{Precision@k} = \frac{R(k)}{S}, \text{Recall@k} = \frac{R(k)}{A} \tag{1}$$

where:

- R(k) represents the number of argument requests where at least one of the top-k recommendations matches the expected argument type.
- S denotes the total number of argument requests in the test set for which the tool supports the expression type of the expected argument.
- A is the total number of argument requests in the test set.

1.2 RQ1:Improving the State of the Art

To address RQ1, we rigorously evaluated `APICopilot` against existing state-of-the-art approaches of API argument completion tasks. The evaluation results are presented in Table 1. The first column categorizes the datasets, specifically "Java" and "Python." The second column specifies the evaluation metrics, "Top-1," "Top-3," and "Top-10," each representing the performance at different levels of recommendation depth. Columns 3-16 show Precision (P) and Recall (R) for models (APICopilot, ARist, CodeT5+, UniXcoder, ChatGPT, Gemini, Llama), categorized by dataset (Java, Python).

Table 1 demonstrates that `APICopilot` consistently outperforms ARist across all Top-K levels in the Java dataset. For the Java dataset, `APICopilot` shows significant improvements at Top-1, achieving a 30.31%=((73.65%-56.52%)/56.52%) relative improvement in P and a 28.27%=((70.91%-55.28%)/55.28%) relative improvement in R. This advantage persists at Top-3, with improvements of 26.48%=((90.83%-71.81%)/71.81%) in P and 25.34%=((87.49%-69.8%)/69.8%) in R. At Top-10, the improvements are 22.50%=((97.13%-79.29%)/79.29%) in P and 21.45%=((93.93%-77.35%)/77.35%) in R.

Table 1 shows that `APICopilot` also outperforms code completion models (CodeT5+, UniXcoder) across all Top-K metrics (P/R) in both Java and Python, with improvements exceeding 20-30%. Compared to LLM-based few-shot learning (ChatGPT, Gemini, Llama), APICopilot is competitive, especially at Top-1 and Top-3, showing smaller performance gaps (e.g., within 1-10%). While LLMs excel at Top-10, APICopilot remains a strong performer at lower Top-K levels. This consistent performance across diverse models and evaluation metrics demonstrates that `APICopilot` significantly advances argument recommendation, consistently outperforming traditional task-specific approaches and LLM-based few-shot learning across programming languages.

1.3 RQ2: Ablation Study

1.3.1 Impact of Set Size of Best Examples. The evaluation results are presented in Table 2. The first column categorizes the datasets, while the second lists evaluation metrics P(%) and R(%), both as percentages. Columns 3-7 show the performance of the model using varying sizes of the "Top-K" best examples across both datasets.

Table 2 shows that increasing the size of K from lower values to higher values generally leads to improvements in both P and R. For the Java dataset, moving from K=30 to K=100 results in a 5.35% relative improvement in P (from 69.91% to 73.65%) and a 5.34% relative improvement in R (from 74.92% to 78.92%). Similarly, in the Python dataset, increasing K from 30 to 100 yields a 4.68% relative improvement in P (from 71.55% to 74.90%) and a 4.68% relative improvement in R (from 76.43% to 80.01%). Notably, the improvements become marginal beyond K=100, indicating a point of diminishing returns. This suggests that while a larger context window (higher K) initially provides more relevant information, beyond K=100, it yields marginal improvements.

1.3.2 Impact of Graph Matching. Table 3 clearly demonstrates the positive impact of incorporating graph matching (GM) into the GraphRAG model. In both Java and Python datasets, the introduction of graph matching significantly enhances performance. Specifically, "GraphRAG + GM-Iso," which utilizes isomorphism-based graph matching, shows improvements over the baseline "GraphRAG (No GM)." For Java, this configuration achieves a 6.46% relative improvement in P and a 6.07% relative improvement in R. Similarly, for Python, the improvements are 6.42% in P and 8.61% in R.

However, the most notable performance gains are observed with "GraphRAG + GM-Full," which employs a more comprehensive graph-matching strategy. This configuration outperforms both the baseline and "GraphRAG + GM-Iso." In the Java dataset, "GraphRAG + GM-Full" achieves a 10.31% relative improvement in P and an 11.58% relative improvement in R compared to the baseline. For Python, the improvements are 10.21% in P and 13.94% in R. Furthermore, comparing "GraphRAG + GM-Full" to "GraphRAG + GM-Iso"

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Table 1: Improving the State of the Art

Types of Approaches		In Context Learning		Task Specific Approach		Code Completion Models				LLM-Based (Few-Shot) Learning					
Datasets	**Metrics**	**APICopilot**		**ARist**		**CodeT5+**		**UniXcoder**		**ChatGPT**		**Gemini**		**Llama**	
		P(%)	**R(%)**	**P(%)**	**R(%)**	**P(%)**	**R(%)**	**P(%)**	**R(%)**	**P(%)**	**R(%)**	**P(%)**	**R(%)**	**P(%)**	**R(%)**
Java	Top-1	73.65	70.91	56.52	55.28	58.68	56.49	59.84	57.62	62.90	60.69	60.24	58.01	62.58	60.25
	Top-3	90.83	87.49	71.81	69.8	71.21	68.58	73.42	70.72	82.01	78.99	79.29	76.38	81.57	78.57
	Top-10	97.13	93.93	79.29	77.35	76.29	73.77	83.12	80.44	87.16	84.38	83.16	80.48	84.74	81.95
Python	Top-1	74.90	69.71	-	-	57.57	53.58	59.44	55.33	64.50	60.02	61.98	57.69	63.75	59.33
	Top-3	92.34	86.01	-	-	71.52	66.61	74.74	69.61	82.57	76.91	79.62	74.22	80.63	75.11
	Top-10	98.78	92.34	-	-	79.15	74.00	83.37	77.95	89.08	83.23	84.07	78.59	86.10	80.49

Table 2: Impact of Size of Top-K Best Examples

Datasets	Metrics	K=30	K=50	**K=100**	K=150	K=200
Java	P(%)	69.91	72.71	**73.65**	73.67	73.69
	R(%)	74.92	77.92	**78.92**	78.94	78.96
Python	P(%)	71.55	73.88	**74.90**	74.92	74.94
	R(%)	76.43	78.91	**80.01**	80.03	80.05

Table 3: Impact of Graph Matching

Configuration	Java		Python	
	P(%)	**R(%)**	**P(%)**	**R(%)**
GraphRAG (No GM)	66.84	63.55	67.96	61.18
GraphRAG + GM-Iso	71.16	67.35	72.32	66.45
GraphRAG + GM-Full	**73.73**	**70.91**	**74.90**	**69.71**
GM-Iso over GraphRAG	6.46	6.07	6.42	8.61
GM-Full over GraphRAG	**10.31**	**11.58**	**10.21**	**13.94**
GM-Full over GM-Iso	3.61	5.29	3.57	4.91

Table 4: Impact of Set Size of Top-K Subgraphs from GM

Datasets	Metrics	K=3	K=5	**K=10**	K=15	K=20
Java	P(%)	72.84	73.38	**73.65**	73.63	73.14
	R(%)	78.05	78.63	**78.92**	78.90	78.37
Python	P(%)	73.58	74.45	**74.90**	74.88	74.39
	R(%)	78.60	79.53	**80.01**	79.99	79.46

reveals that the full graph matching approach provides additional gains, with improvements of 3.61% in P and 5.29% in R for Java, and 3.57% in P and 4.91% in R for Python. These results underscore the importance of graph matching in enhancing the performance of GraphRAG. The full graph matching approach, in particular, demonstrates a substantial ability to improve both P and R, indicating that a more comprehensive comparison of graph structures leads to more accurate and complete recommendations. This highlights the effectiveness of leveraging graph-based representations for API argument completion tasks.

1.3.3 Impact of Set Size of Top-k Subgraphs. Table 4 shows that adjusting the size of K, representing the number of top-K subgraphs considered from graph matching, impacts performance. For the Java dataset, increasing K from 3 to 10 results in a 1.11% relative improvement in P (from 72.84% to 73.65%) and a 1.11% relative improvement in R (from 78.05% to 78.92%). Similarly, in the Python dataset, increasing K from 3 to 10 yields a 1.79% relative improvement in P (from 73.58% to 74.90%) and a 1.79% relative improvement in R (from 78.60% to 80.01%). However, the performance declines slightly beyond K=10, indicating an optimal set size. This suggests that while a larger set of subgraphs initially provides more relevant information, an excessive number can introduce noise or redundancy, negatively affecting performance. The peak performance at K=10 highlights the importance of selecting an appropriate set size for subgraph consideration in graph matching.

1.4 RQ3: Working with Various LLMs

Table 6 presents the performance of various LLMs both in their native few-shot setting and when enhanced by `APICopilot`, highlighting the latter's augmentation capabilities. Integrating `APICopilot` consistently and significantly improves the performance of all tested LLMs (ChatGPT-4o, Gemini-1.5, Llama-3) across both Java and Python datasets. Specifically, for ChatGPT-4o, P and R improvements are 17.09% and 16.84% for Java, and 16.12% and 16.14% for Python, respectively. For Gemini-1.5, improvements reach 20.44% and 20.41% for Java and 19.04% and 19.03% for Python. Llama-3 also benefits, with improvements of 13.68% and 18.07% for Java and 14.89% and 23.45% for Python. These substantial and consistent gains across diverse LLMs and datasets underscore `APICopilot`'s effectiveness and broad applicability in enhancing the quality of LLM-based argument recommendations for code completion.

Table 5: Top-1 Performance: Seen vs. Unseen Java Data

Types of Approaches		In Context Learning		Task Specific Approach		Code Completion Models				LLM-Based (Few-Shot) Learning					
Datasets	**Metrics**	**APICopilot**		**ARist**		**CodeT5+**		**UniXcoder**		**ChatGPT**		**Gemini**		**Llama**	
		P(%)	**R(%)**	**P(%)**	**R(%)**	**P(%)**	**R(%)**	**P(%)**	**R(%)**	**P(%)**	**R(%)**	**P(%)**	**R(%)**	**P(%)**	**R(%)**
Seen	Top-1	73.65	70.91	56.52	55.28	58.68	56.49	59.84	57.62	62.90	60.69	60.24	58.01	62.58	60.25
Unseen	Top-1	72.56	69.82	51.71	50.59	51.93	48.59	52.96	49.91	55.04	52.60	52.71	49.96	56.32	53.63
Drop (-%)		1.48	1.54	8.51	8.48	11.51	13.98	11.49	13.38	12.50	13.33	12.50	13.88	10.00	10.99

Table 6: Working with Various LLMs

		Java		**Python**	
LLM	**Setting**	**P(%)**	**R(%)**	**P(%)**	**R(%)**
ChatGPT-4o	Few-Shot	62.90	60.69	64.50	60.02
	`APICopilot`	73.65	70.91	74.90	69.71
	Improvement	**17.09**	**16.84**	**16.12**	**16.14**
Gemini-1.5	Few-Shot	60.24	58.01	61.98	57.69
	`APICopilot`	72.55	69.85	73.78	68.67
	Improvement	**20.44**	**20.41**	**19.04**	**19.03**
Llama-3	Few-Shot	62.58	60.25	63.75	59.33
	`APICopilot`	71.14	71.14	73.24	73.24
	Improvement	**13.68**	**18.07**	**14.89**	**23.45**

1.5 RQ4: Performance on Unseen Project

Table 5 shows that all models experience performance drops on 'Unseen' data. APICopilot demonstrates superior robustness to domain shifts with the smallest decline (1.48% P, 1.54% R), indicating strong generalization, while ARist, CodeT5+, and UniXcoder show larger drops (8-14%), as do LLMs (10-13%).