A Systematic Evaluation of Large Code Models in API Suggestion: When, Which, and How

Chaozheng Wang
The Chinese University of Hong Kong
Hong Kong, China
adf111178@gmail.com

Shuzheng Gao
The Chinese University of Hong Kong
Hong Kong, China
szgao23@cse.cuhk.edu.hk

Cuiyun Gao*
Harbin Institute of Technology
Shenzhen, China
gaocuiyun@hit.edu.cn

Wenxuan Wang
The Chinese University of Hong Kong
Hong Kong, China
wxwang@cse.cuhk.edu.hk

Chun Yong Chong
Huawei
Hong Kong, China
chunyong@ieee.org

Shan Gao
Huawei
Shenzhen, China
gaoshan17@huawei.com

Michael R. Lyu
The Chinese University of Hong Kong
Hong Kong, China
lyu@cse.cuhk.edu.hk

ABSTRACT

API suggestion is a critical task in modern software development, assisting programmers by predicting and recommending third-party APIs based on the current context. Recent advancements in large code models (LCMs) have shown promise in the API suggestion task. However, they mainly focus on suggesting which APIs to use, ignoring that programmers may demand more assistance while using APIs in practice including when to use the suggested APIs and how to use the APIs. To mitigate the gap, we conduct a systematic evaluation of LCMs for the API suggestion task in the paper.

To facilitate our investigation, we first build a benchmark that contains a diverse collection of code snippets, covering 176 APIs used in 853 popular Java projects. Three distinct scenarios in the API suggestion task are then considered for evaluation, including (1) "*when to use*", which aims at determining the desired position and timing for API usage; (2) "*which to use*", which aims at identifying the appropriate API from a given library; and (3) "*how to use*", which aims at predicting the arguments for a given API. The consideration of the three scenarios allows for a comprehensive assessment of LCMs' capabilities in suggesting APIs for developers. During the evaluation, we choose nine popular LCMs with varying model sizes for the three scenarios. We also perform an in-depth analysis of the influence of context selection on the model performance. Our experimental results reveal multiple key findings. For instance, LCMs present the best performance in the "how to use" scenario while performing the worst in the "when to use" scenario, e.g., the average performance gap of LCMs between "when to use" and "how to use" scenarios achieves 34%, indicating that the "when to use" scenario is more challenging. Furthermore, enriching context information substantially improves the model performance. Specifically, by incorporating the contexts, smaller-sized LCMs can outperform those twenty times larger models without the contexts provided. Based on these findings, we finally provide insights and implications for researchers and developers, which can lay the groundwork for future advancements in the API suggestion task.

CCS CONCEPTS

• **Software and its engineering → Software development techniques**;

KEYWORDS

large code models, API suggestion, empirical study

ACM Reference Format:
Chaozheng Wang, Shuzheng Gao, Cuiyun Gao, Wenxuan Wang, Chun Yong Chong, Shan Gao, and Michael R. Lyu. 2024. A Systematic Evaluation of Large Code Models in API Suggestion: When, Which, and How. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695004

1 INTRODUCTION

API suggestion is a critical task in modern software development, aiming to assist programmers by predicting and recommending third-party API usage based on the current context [5, 29, 40]. With the development of deep learning, multiple techniques have been proposed to provide intelligence API suggestions. In recent years, the emergence of large language models (LLMs) has revolutionized various natural language processing (NLP) tasks [4, 37, 43]. Inspired by their success, researchers have adapted these models to the domain of programming languages, giving rise to large code models (LCMs) [16, 20, 22, 33]. LCMs have shown remarkable improvements in the API suggestion task [6, 27], by leveraging their ability

*Corresponding author. The author is also affiliated with Peng Cheng Laboratory.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695004

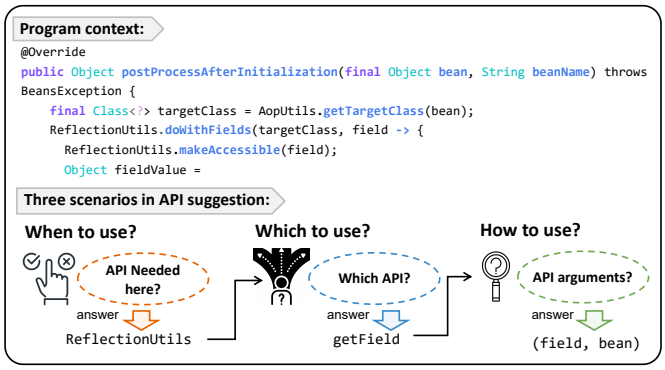


Figure 1: Three distinct scenarios in the API suggestion task.

to capture complex patterns and semantics from vast amounts of source code corpus.

Despite the impressive performance of LCMs in the API suggestion task, the previous studies mainly focus on suggesting appropriate APIs from a given library to use [5, 6], which do not involve the common API usage practices faced by developers as illustrated in Figure 1. Given the convenience and powerful functionalities of third-party APIs, developers can improve their programming efficiency and productivity. However, the huge amount of APIs requires great effort to memorize. Specifically, developers may demand more assistance while using APIs in practice including when to use APIs and how to use APIs. However, these scenarios of the API suggestion task remain unexplored, leaving a substantial gap in understanding and supporting the diverse needs of developers in real-world programming environments.

In this paper, we present a systematic evaluation of LCMs in API suggestion tasks, aiming at addressing the critical gap in the existing literature. To facilitate the investigation, we first build a benchmark for covering the various API suggestion scenarios. Our benchmark comprises a collection of 4,146 entries of API usage in 3,136 code files, covering 176 diverse APIs sourced from 853 popular Java projects. By curating a wide range of API usage patterns and contexts, we aim to provide a representative and diverse evaluation dataset for assessing the capabilities of LCMs in real-world API suggestion tasks.

To achieve a comprehensive evaluation, we propose to divide the API suggestion task into three distinct scenarios to simulate the practices of developers: (1) "*when to use APIs*", which aims to determine to use APIs in appropriate positions; (2) "*which API to use*", which aims to identify the desired API from the given library; and (3) "*how to use APIs*", aiming at predicting the arguments for a given API (illustrated in Figure 1). This task decomposition allows us to assess the LCMs' ability to understand and generate code in different API usage scenarios. Furthermore, considering that different contexts provide varying levels of information and guidance, we investigate the impact of various contexts on the performance of LCMs by incorporating different aspects of contexts such as function comments and import messages as shown in Figure 2. By systematically varying the amount and type of contextual information provided to the LCMs, we aim to identify the most effective prompting strategies for the API suggestion task. To ensure a thorough exploration of the capabilities of LCMs, we include nine LCMs in our experiments: StarCoder [22], CodeLlama [33], and DeepSeek-Coder [16]. These models encompass a wide range of sizes, i.e., from 1 billion to 34 billion parameters, which also enables us to examine the impact of model scales on performance.

Based on our experimental results, we achieve the following key findings: (1) **LCMs present the best performance in the "how to use" scenario while performing the worst in the "when to use" scenario.** Specifically, the average performance gap of LCMs between "when to use" and "how to use" scenarios achieves 34%. ; (2) **Enriching context information can substantially improve the performance of the API suggestion task.** For instance, we find that including file contexts, function comments, import messages, and suffix contexts in the input prompt, the average exact match score in the "when to use" scenario increases by 89.2%. Specifically, by incorporating additional contexts, smaller-sized LCMs (e.g., DeepSeek-Coder 1.3B) can outperform those twenty times larger models (e.g., DeepSeek-Coder 33B) without the contexts provided. (3) **Enriching context information will increase the number of tokens input to LCMs, which consequently decreases the model throughput.** For instance, the average number of tokens will increase more than three times after incorporating all studied contexts, while the average throughput of LCMs drops by 54%. Based on the findings, we finally provide valuable insights into the strengths and limitations of current LCMs in handling API suggestion tasks, which can lay the groundwork for future advancements in the API suggestion task.

Our main contributions can be summarized as follows:

- To the best of our knowledge, we are the first to divide the API suggestion task into three scenarios including "how to use", "which to use", and "when to use". The categorization fills a critical gap in the existing literature.
- We propose an API suggestion benchmark specifically designed to assess the capabilities of LCMs in real-world API usage scenarios. Based on the benchmark, we conduct extensive experiments with nine popular LCMs for the three scenarios.
- Based on the results, we finally provide insights and implications for researchers and developers, which can lay the groundwork for future advancements in the API suggestion task.
- To foster reproducibility and encourage further research in this area, we make our data and code publicly available at *https://github.com/adf1178/api_suggestion_evaluation*.

2 OVERVIEW OF METHODOLOGY

In this section, we introduce our evaluation methodology from benchmark preparation, context types, and research questions, respectively.

2.1 Benchmark Preparation

In this study, we focus exclusively on APIs from the Spring Framework for several key reasons. SpringFramework is one of the most popular frameworks in web development, which is widely used in previous studies [10]. Furthermore, it is highly prevalent in Maven

Figure 2: The illustration of different context types.

repositories [1]. Its widespread adoption ensures that our benchmark is both relevant and representative of real-world API usage scenarios. In addition, its extensive documentation provides clear and complete descriptions of the APIs, which is essential for comprehensively evaluating the performance of LCMs in API suggestion tasks. By concentrating on a single and well-established framework with thorough documentation, we can conduct a more focused and in-depth analysis.

2.1.1 Data Collection. First, we obtain high-star repositories from GitHub by selecting those with more than 100 stars. We then filter out the repositories that do not utilize the Spring Framework, resulting in a final set of 853 projects. Second, for each file in these projects, we use the Tree-sitter [38] to construct an abstract syntax tree (AST). By traversing the nodes of the AST, we identify function calls and determine if these calls correspond to APIs from the Spring Framework. Third, based on the frequency of API calls, we prioritize APIs with the highest usage, retaining those that are used more than ten times. This process yields a set of 176 APIs. We then retain the files within the projects that use these high-frequency APIs. To ensure diversity, we select only one file per project for each API, ensuring a varied dataset. Finally, our benchmark contains 3,136 code files and 4,146 API usage entries.

2.1.2 Benchmark Construction. After collecting API data, we categorize the API suggestion scenarios into three scenarios including "*when to use*", "*which to use*", and "*how to use*" as shown in Figure 1. These scenarios simulate different situations where developers use APIs in programming practice and evaluate the capabilities of current LCMs to generate accurate suggestions in these scenarios.

When to Use API. In the "when to use" scenario, we evaluate the capabilities of LCMs to correctly call APIs based on the surrounding code structure and logic. This scenario simulates the situation where developers may be aware of the available APIs but struggle with determining the appropriate timing or location to invoke them within the code. Taking Figure 1 as an example, based on the surrounding code, the model should determine that "*ReflectionUtils.getField(field, bean)*" is the correct API call to use and place it in the designated location.

Which API to Use. Which API to use, also called API recommendation in existing studies [5], reflects the scenario in which developers input the parent library and the *dot* operator and expect code completion tools to predict the exact API to use. For instance, give "*ReflectionUtils.*" and expect LCMs to suggest "*getField(field, bean)*". The "which API to use" scenario arises when developers have specific functionality in mind but are unsure about the most appropriate API to achieve their goal.

How to use APIs. In this scenario, we simulate the situation where developers input the API and expect LCMs to predict the arguments of the API (API arguments prediction) as illustrated in Figure 1. Due to the increasing number of third-party libraries and their corresponding APIs, developers may struggle to remember how to use the APIs in detail.

2.2 Context Types

Based on the collected data on API suggestion, we further categorize the contexts in the code file to quantitatively evaluate the influence of different types of contexts on model performance. Specifically, we utilize the following types of contexts which are demonstrated in Figure 2 to construct the prompt fed into LCMs and evaluate their performance.

- **Function Context (Base)** is the fundamental context used to feed into LCMs for suggesting API usage. Specifically, we select the source code from the function signature of the function involving the target API up to the target API itself. We use function context in our experiments because it represents the basic unit of API usage.
- **File Context (F)** represents the source code outside the function that involves the target API. In addition to the function context, considering the code contexts at the file level provides LCMs with a broader context, potentially improving their performance in providing API suggestions. It is important to note that a file may contain a substantial amount of

source code, which can exceed the context limit of LCMs and result in notable time consumption. Therefore, we select k lines of code preceding the function to construct the file context.

- **Function Comment (C)** typically describes the purpose and logic of the corresponding functions [11, 14]. We incorporate function comments into the context to examine whether explaining the function's utilities in natural language can enhance the performance of LCMs in API suggestion tasks.
- **Suffix Context (S)** refers to the source code following the called API statement. This consideration arises from the fact that developers often edit existing code rather than always appending new code at the end. While current LCM architectures typically generate code tokens auto-regressively (i.e., one token at a time), researchers have proposed fill-in-the-middle (FIM) tasks for pre-training. Through FIM, LCMs can incorporate suffix context to complete intermediate code segments, potentially enriching the model input and enhancing performance in API suggestion tasks.
- **Import Messages (I)** contain the libraries imported in the file and indicate what APIs can be called, which motivates us to experiment with this kind of context as the LCMs' prompt.
- **Library Candidates (L)** provide all of the usable APIs in the currently used parent library, which are designed for the "which to use" scenario particularly. We construct this context due to the hallucination issues of LLMs [32]. In API suggestion scenarios, LCMs may fabricate some APIs that do not exist in the library. Thus, we explore whether explicitly providing usable APIs to LCMs can improve their performance.

2.3 Research Questions

2.3.1 RQ1: How do different LCMs perform in the three scenarios of API suggestion? In this research question, we divide the API suggestion task into three scenarios, evaluating and comparing the performance of LCMs in each scenario given basic function contexts.

2.3.2 RQ2: How different types of contexts affect LCMs performance in API suggestion? In this research question, we investigate the influence of involving different types of contexts on the model performance in API suggestion. Specifically, we utilize three sub-research questions to investigate the influence of the three scenarios, respectively.

2.3.3 RQ3: How do contexts affect the token length and throughput of LCMs? More contexts can enrich the semantics of prompts and provide more information but may increase the length of input tokens, which potentially brings overhead to LCMs' inference. Thus, in this RQ, we explore the influence of different types of contexts on the prompt length and throughput of LCMs.

3 EXPERIMENT SETUP

3.1 Selected LCMs

In this paper, we select three kinds of popular and state-of-the-art LCMs with their versions in different sizes. In specific, our selected LCMs are:

- **StarCoder** [20] is a large language model trained on the mixture of source code and natural language texts. Its training data incorporate more than 80 different programming languages as well as text extracted from GitHub issues and commits and from notebooks. We select its 3B, 7B, and 15B versions in our experiments.
- **CodeLlama** [33] is a family of large language models for code based on LLama 2 [37] with state-of-the-art code generation, blank infilling, and long-context processing capabilities. In this paper, we choose CodeLlama's base model (i.e., CodeLlama Base) in three different sizes including 7B, 13B, and 34B for instruction tuning.
- **DeepSeek-Coder** [16] is a series of large code models that have an identical architecture to CodeLlama. DeepSeek-Coder is trained from 2T tokens from scratch. Specifically, we choose DeepSeek-Coder Base in sizes of 1.3B, 6.7B, and 33B in this paper.

3.2 Evaluation Metrics

In this paper, following previous studies [23, 27, 35, 36], we utilize three metrics to evaluate the performance of LCMs in the API suggestion tasks including exact match (EM), API usage accuracy, and edit similarity, respectively.

3.2.1 Exact Match. The exact match metric measures whether the model output is the same as the ground truth, which is the most strict metric.

3.2.2 API Usage Accuracy. API usage accuracy is utilized to evaluate whether LCMs can predict the desired API in "which to use" and "when to use" scenarios.

3.2.3 Edit Similarity. The edit similarity metric is used to measure how closely the model's output resembles the ground truth, considering the edits required to transform one into the other.

3.3 Implementation Details

All the experiments are run on a server with 2*A100 GPUs with 80GB graphic memory. For fast inference, we utilize vLLM [19] based on PagedAttention to improve efficiency. To eliminate the influence of random sampling, we utilize greedy decoding strategy during inference. In addition, we employ the Flash-Attention technique [8] for long-context optimization.

4 EXPERIMENT RESULTS

4.1 RQ1: Model Performance in API Suggestion

We present the results of nine LCMs, given the basic function context, in the three scenarios of API suggestion in Table 1. From the table, we achieve the following observations.

(1) The model performance increases as the complexity of the scenarios decreases. For the three scenarios studied in this paper, the "when to use" scenario is the most challenging, and "how to use" is the simplest one. We estimate the difficulties of different scenarios based on the fact that "how to use" only requires models to predict the API arguments, while the "which to use" and "when to use" scenarios require further prediction of the specific API (e.g., *"getField"*) and its library (e.g., *"ReflectionUtils"*), respectively.

Table 1: Results of different LCMs in three scenarios of API Suggestion.

Metrics	SC-3B	SC-7B	SC-15B	CL-7B	CL-13B	CL-34B	DSC-1.3B	DSC-6.7B	DSC-33B	Avg
When to use										
Exact Match	25.22	30.49	31.79	30.84	34.54	31.52	23.64	30.62	**35.22**	30.43
API Acc.	35.53	42.22	43.22	42.95	46.85	43.99	34.13	42.15	**47.71**	42.08
Edit Sim	59.77	64.17	64.83	64.63	66.93	65.03	58.89	63.42	**67.32**	63.89
Which to use										
Exact Match	50.96	55.07	56.26	54.40	57.20	53.95	46.14	53.79	**57.22**	53.89
API Acc.	77.06	80.08	81.49	79.47	**81.83**	78.77	71.15	78.31	81.29	78.82
Edit Sim	81.77	83.46	84.11	83.25	**84.33**	82.80	79.28	83.07	84.12	82.91
How to use										
Exact Match	61.62	64.66	65.34	64.55	**66.24**	64.69	59.23	64.21	65.85	64.04
Edit Sim	84.72	86.27	86.50	86.15	**86.88**	85.91	83.34	86.06	86.69	85.83

Table 2: Results in the "when to use" scenario. SC, CL, and DSC indicate StarCoder, CodeLlama, and DeepSeek-Coder, respectively.

Method	SC-3B	SC-7B	SC-15B	CL-7B	CL-13B	CL-34B	DSC-1.3B	DSC-6.7B	DSC-33B	Avg	Improve
Exact Match											
Base	25.22	30.49	31.79	30.84	34.54	31.52	23.64	30.62	35.22	30.43	
Base+F	27.82	32.62	34.71	33.18	37.65	34.04	25.77	33.39	38.49	33.07	(↑ 8.7%)
Base+F+C	34.90	40.09	41.96	41.30	45.41	41.73	33.05	42.51	46.03	40.77	(↑ 34.0%)
Base+F+S	36.11	41.75	43.75	41.37	45.89	45.12	33.25	42.83	48.56	42.07	(↑ 38.2%)
Base+F+I	38.36	44.21	48.08	45.76	49.12	47.31	37.51	46.07	50.82	45.25	(↑ 48.7%)
Base+F+C+I	43.45	49.39	52.98	50.40	53.40	51.95	43.22	51.15	55.64	50.17	(↑ 64.9%)
Base+F+C+I+S	**52.15**	**58.03**	**58.68**	**56.62**	**59.88**	**59.33**	**49.42**	**59.94**	**64.08**	**57.57**	(↑ 89.2%)
API Usage Accuracy											
Base	35.53	42.22	43.22	42.95	46.85	43.99	34.13	42.15	47.71	42.08	
Base+F	38.22	44.51	46.78	45.63	50.46	46.56	36.28	45.13	50.69	44.92	(↑ 6.9%)
Base+F+C	47.58	51.79	56.51	56.09	60.41	57.69	46.07	56.96	61.18	54.92	(↑ 30.7%)
Base+F+S	46.76	53.58	55.35	53.26	65.74	64.99	43.82	54.26	60.68	55.38	(↑ 31.8%)
Base+F+I	54.58	62.57	64.27	62.67	65.74	64.57	53.12	62.27	67.52	61.92	(↑ 47.3%)
Base+F+C+I	60.65	66.74	69.89	68.02	70.83	69.87	59.79	68.27	72.92	67.44	(↑ 60.4%)
Base+F+C+I+S	**67.38**	**73.03**	**73.69**	**72.92**	**76.39**	**75.89**	**65.38**	**75.35**	**79.42**	**73.27**	(↑ 74.3%)
Edit Similarity											
Base	59.77	64.17	64.83	64.63	66.93	65.03	58.89	63.42	67.32	63.89	
Base+F	61.75	65.61	67.18	66.26	69.39	65.44	60.48	65.75	69.19	65.67	(↑ 2.8%)
Base+F+C	67.09	70.60	72.21	71.71	74.96	72.78	66.24	72.09	74.81	71.39	(↑ 11.7%)
Base+F+S	68.64	72.50	73.91	72.54	75.94	75.07	67.11	73.14	76.78	72.85	(↑ 14.0%)
Base+F+I	70.71	74.79	76.64	75.94	77.61	76.99	70.08	75.10	78.43	75.14	(↑ 17.6%)
Base+F+C+I	73.95	77.56	79.41	78.77	80.29	79.52	73.85	78.47	81.18	78.11	(↑ 22.3%)
Base+F+C+I+S	**79.69**	**83.15**	**83.58**	**83.14**	**85.15**	**84.11**	**78.84**	**84.24**	**86.49**	**83.15**	(↑ 30.1%)

From table 1, we find that in the most challenging "when to use" scenario, which requires LCMs to determine whether to use APIs by themselves, the models achieve an average exact match rate of 30.43. For the relatively simpler "which to use" and "how to use" scenarios, the average exact match scores for nine LCMs increase to 53.89 and 64.04, representing the improvement of 77% and 110%, respectively.

(2) Model performance is positively correlated to the model sizes, and the correlation is more pronounced in more challenging scenarios. During evaluating the performance of various LCMs, we observe a clear positive correlation between model sizes and performance. For instance, in the "when to use" scenario, the Pearson correlation coefficient between model size and the exact match score achieves 0.67 with a p-value of 0.049, indicating a significant relationship. This positive correlation can be attributed to larger models' enhanced capacity to understand context, resulting in more accurate completions. However, it is important to note that larger models do not consistently outperform their smaller counterparts. Among the tested LCMs with basic context, DeepSeek-Coder with 33B parameters achieves the highest performance in the "when to use" scenario, and CodeLlama 13B performs the best in the "which to use" and "how to use" scenarios.

As the complexity of scenarios decreases, the relationship becomes less pronounced accordingly. Specifically, the correlation coefficients between model sizes and exact match score decrease to 0.53 and 0.32 in the "which to use" and "how to use" scenarios,

Table 3: Results in "which to use" scenario. SC, CL, and DSC indicate StarCoder, CodeLlama, and DeepSeek-Coder, respectively.

Method	SC-3B	SC-7B	SC-15B	CL-7B	CL-13B	CL-34B	DSC-1.3B	DSC-6.7B	DSC-33B	Avg	Improve
Exact Match											
Base	50.96	55.07	56.26	54.40	57.20	53.95	46.14	53.79	57.22	53.89	
Base+F	53.18	56.67	58.34	57.09	59.40	56.42	48.43	56.08	59.49	56.12	(↑ 4.1%)
Base+F+L	53.09	57.33	58.86	57.65	59.68	56.59	49.61	57.44	59.74	56.67	(↑ 5.2%)
Base+F+C	58.00	61.46	62.33	61.52	63.59	61.05	54.45	61.37	64.14	60.88	(↑ 13.0%)
Base+F+S	60.26	63.32	65.48	61.49	64.42	62.50	50.69	60.49	65.86	61.61	(↑ 14.3%)
Base+F+I	55.25	60.05	62.12	59.63	62.96	61.22	53.41	60.12	63.08	59.76	(↑ 10.9%)
Base+F+C+I	58.84	63.69	65.53	63.91	66.36	64.86	58.05	64.61	67.53	63.71	(↑ 18.2%)
Base+F+C+I+S	**66.40**	**69.92**	**71.77**	**67.53**	**70.26**	**68.69**	**59.11**	**68.19**	**72.69**	**68.28**	(↑ 26.7%)
Base+F+L+C+I+S	64.85	68.46	71.22	66.64	69.55	68.21	58.24	67.51	71.94	67.40	(↑ 25.1%)
API Usage Accuracy											
Base	77.06	80.08	81.49	79.47	81.83	78.77	71.15	78.31	81.29	78.82	
Base+F	79.00	81.35	82.62	80.84	83.31	80.91	73.18	80.07	82.91	80.46	(↑ 2.1%)
Base+F+L	79.34	82.26	83.85	81.89	83.66	82.07	75.22	82.56	83.72	81.62	(↑ 3.5%)
Base+F+C	84.49	86.43	87.18	86.03	87.92	85.86	80.32	86.15	87.96	85.81	(↑ 8.9%)
Base+F+S	81.17	82.53	85.11	82.16	86.46	84.75	71.56	80.68	85.79	82.24	(↑ 4.3%)
Base+F+I	82.12	84.90	85.82	83.64	86.46	84.77	79.03	84.14	85.90	84.08	(↑ 6.7%)
Base+F+C+I	85.54	88.31	89.39	87.58	89.56	88.20	83.34	87.94	90.05	87.77	(↑ 11.3%)
Base+F+C+I+S	**87.20**	**88.90**	**90.52**	**87.66**	**89.73**	**89.11**	**81.68**	**88.07**	**91.29**	**88.24**	(↑ 12.0%)
Base+F+L+C+I+S	84.68	86.71	89.24	86.54	88.75	88.69	80.82	87.75	90.86	87.12	(↑ 10.5%)
Edit Similarity											
Base	81.77	83.46	84.11	83.25	84.33	82.80	79.28	83.07	84.12	82.91	
Base+F	83.07	84.45	85.15	84.59	85.38	84.28	80.51	84.18	85.40	84.12	(↑ 1.5%)
Base+F+L	82.99	84.72	85.42	84.96	85.65	84.64	81.21	84.89	85.72	84.47	(↑ 1.9%)
Base+F+C	85.33	86.45	87.09	86.67	87.47	86.48	83.59	86.73	87.66	86.39	(↑ 4.2%)
Base+F+S	85.74	86.54	87.68	86.44	87.40	86.61	81.53	85.90	88.08	86.21	(↑ 4.0%)
Base+F+I	84.60	86.13	86.90	86.11	87.16	86.52	83.35	86.14	87.31	86.02	(↑ 3.7%)
Base+F+C+I	86.14	87.68	88.52	87.88	88.75	88.16	85.28	87.96	89.22	87.73	(↑ 5.8%)
Base+F+C+I+S	**88.62**	**89.70**	**90.42**	**89.23**	**90.04**	**89.74**	**85.58**	**89.44**	**91.05**	**89.31**	(↑ 7.7%)
Base+F+L+C+I+S	87.58	89.19	90.20	88.82	89.87	89.05	85.17	89.29	90.88	88.89	(↑ 7.2%)

respectively. Such a relationship is also reflected in the performance gap between large and small models. For instance, in the "when to use" scenario, DeepSeek-Coder 33B outperforms the 1.3B version by 49% in terms of the exact match score. However, the difference is narrowed to 24% and 11% in the "which to use" and "how to use" scenarios, respectively. Consequently, we conclude that the impact of model size on performance becomes more substantial as the complexity of the scenarios increases.

Finding 1: The model performance increases as the complexity of the scenarios decreases, i.e., the gap of average exact match score between "when to use" and "how to use" scenarios achieves 34%. In addition, model performance is positively correlated to the model sizes, and the correlation is more pronounced in more challenging scenarios.

4.2 RQ2: Influence of Contexts on Effectiveness

In this research question, we investigate the influence of contexts on model performance in each scenario, respectively.

4.2.1 RQ2.1 When to Use. We evaluate LCMs in the "when to use" scenario, with the results shown in Table 2.

From the table, we can find that LCMs achieve an average of 30.43, 42.08, and 63.89 in exact match, API Usage Accuracy, and edit similarity metrics given basic function context (Base), respectively. These results indicate that current LCMs struggle to effectively suggest the use of APIs in the desired positions with only local function contexts. Besides function context, pretending file context (F) contributes to LCMs' performance (Base+F), improving the three metrics by 8.7%, 6.9%, and 2.8%, respectively. Such results suggest that the file context provides additional information that helps LCMs more accurately complete API calls. Note that due to the potential sizes of files, we include ten lines of code before the function as the file context [21]. We further explore the influence of varying amounts of file context on model performance in Section 5.1.

Based on file contexts, we further explore the influence of code comments (+C), import messages (+I), and suffix contexts (+S), respectively. From the table, we observe that these three contexts can further improve the performance of LCMs in API suggestion, i.e., the exact match is increased by 34.0%, 48.7%, and, 38.2% compared to that with basic function context, respectively. We attribute the improvement of adding code comments to that comments reflect the code's functionalities and thus provide more guidance for using the APIs. With import messages that define the parent libraries to use in the file, LCMs can better understand the intent of current files and fill in API arguments that conform to the development

requirements. In addition, the broad generation space in the "when to use" scenario requires LCMs to determine whether to use APIs. Therefore, import messages improve the performance by a large margin, which is the largest improvement among the three kinds of contexts. As observed from the table, the suffix contexts also contribute substantially to the performance, i.e., the exact match is increased by 38.2%. Such improvement can be attributed to that 1): involving the suffix contexts ensures contextual coherence: The suffix context provides the expected direction of the code, allowing the model to better understand the current code's logic and structure, leading to more reasonable completions; and 2) suffix context helps to resolve ambiguities present in the earlier parts of the code, providing clear context and making the model's predictions more precise and consistent. We provide a case study to further demonstrate the influence of suffix contexts in Section 5.2.

Furthermore, we investigate the impact of combining different types of contexts (i.e., rows Base+F+C+I and Base+F+C+I+S). We can observe that involving both comments and import messages further increases the performance of LCMs, i.e., the three evaluation metrics are increased by 64.9%, 60.4%, and 30.1%, respectively. After combining comments, import messages, and suffix contexts with file contexts, LCMs obtain the most performance gain by nearly 90% (the exact match rate is improved from 30.43 to 57.57). In addition, we observe that incorporating all studied contexts, small-sized LCMs can outperform those twenty times larger models with the basic function context. For instance, DeepSeek-Coder 1.3B with Base+F+C+I+S contexts outperforms DeepSeek-Coder 33B with basic function context by 40% in terms of the exact match score.

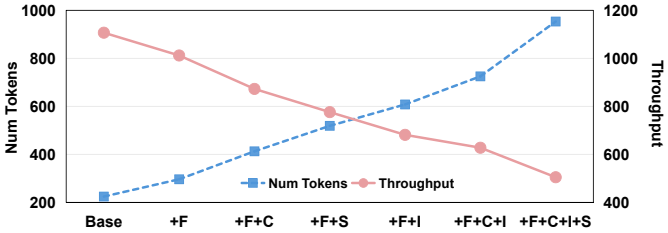
The results demonstrate that different types of contexts contribute variably to API suggestion, and enriching prompts by combining these contexts further enhances the performance of LCMs in suggesting APIs.

4.2.2 RQ2.2: Which to Use. The results of LCMs in the "which to use" scenario of API suggestion are shown in Table 3.

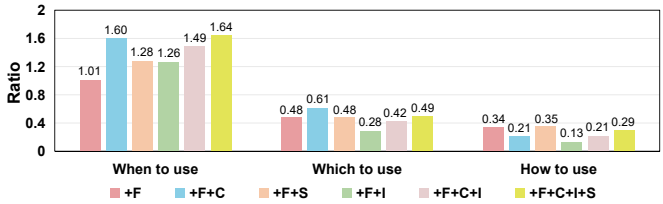
Similarly, the improvement obtained by enriching contexts is also observed in this scenario. For instance, including file contexts (+F) and equipped with additional information such as comments and suffix contexts (+C and +S) present 13% and 14% improvement in terms of the exact match metric, respectively. In addition, we also experiment with a unique type of context in the "which to use" scenario, i.e., library candidates (+L). From the table, we find that incorporating library candidates (+F+L) further improves the performance of file contexts (+F) by 1.1%, 1.4%, and 0.4% on the exact match, API usage accuracy, and edit similarity, respectively. The improvement suggests that informing LCMs of available APIs is helpful in predicting APIs more accurately. However, when involving all of the contexts (+F+L+C+I+S), LCMs perform worse than the combination without library candidates (+F+C+I+S), i.e., 67.40 and 68.28 in the exact match score. The difference indicates that when involving enough contexts, information about library candidates may become redundant.

4.2.3 RQ2.3: How to Use. We investigate how LCMs perform in the "how to use" scenario (i.e., the ability to fill an API's arguments), and the results are presented in Table 4.

Regarding the influence of different contexts, our results indicate that the contribution of contexts in the "how to use" scenario is less



(a) Token numbers and average throughput of different contexts.



(b) Avg. Exact Match Improvement / Avg. Throughput Decrease ratio of different contexts in different scenarios.

Figure 3: Average token length, throughput, and the ratio between performance and throughput.

substantial compared to that in the other scenarios. For instance, after involving all available contexts (+F+C+I+S), the exact match and edit similarity are improved by 15.8% and 5.4% compared with function context, respectively. These results suggest that, despite the naturally higher accuracy in the "how to use" scenario, LCMs derive less additional benefit from the contexts. We hypothesize that in simpler scenarios, LCMs already achieve high accuracy with minimal context, thus additional context provides relatively less guidance and improvement compared to that in more complex scenarios where the LCMs benefit more from enriched contextual information.

Finding 2: Enriching contexts can substantially enhance the performance of the API suggestion task, with improvements becoming more pronounced as the complexity of scenarios increases. Furthermore, equipped with all studied contexts, smaller-sized LCMs can outperform more than twenty times larger models when no additional context is provided.

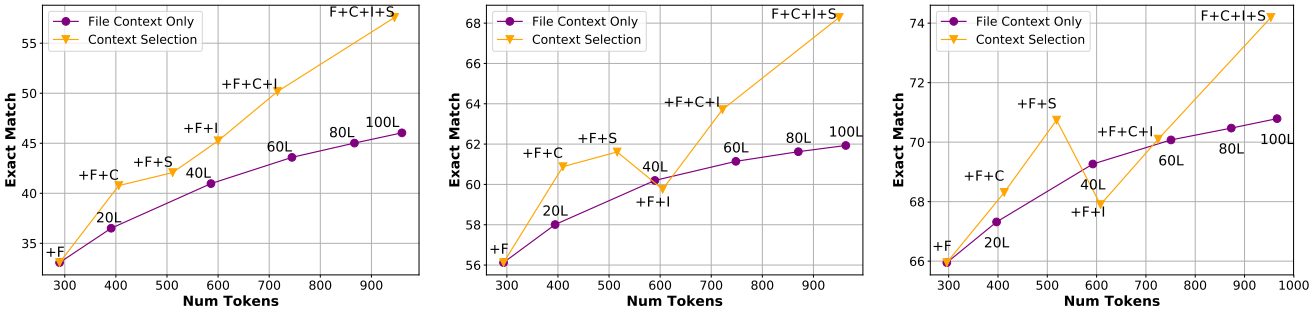
4.3 RQ3: Influence of Contexts on Efficiency

Based on the results in RQ2, we observe that incorporating contexts enhances the model performance. However, they also increase the prompt length, possibly leading to higher latency. To comprehensively investigate the influence of involving different contexts on models' performance, we explore how different contexts influence the lengths of input tokens. In addition, we also explore the model efficiency by recording the model throughput under different context settings. The throughput is defined as the average number of tokens that LCMs generate per second [2, 19, 28]. The results are presented in Figure 3. From the figure, we achieve the following observations.

(1) Enriching contexts increases the prompt length, which consequently decreases the model throughput. Figure 3 (a) presents the average token lengths of input prompts with different

Table 4: Results in the "how to use" scenario. SC, CL, and DSC indicate StarCoder, CodeLlama, and DeepSeek-Coder, respectively.

Method	SC-3B	SC-7B	SC-15B	CL-7B	CL-13B	CL-34B	DSC-1.3B	DSC-6.7B	DSC-33B	Avg	Improve
Exact Match											
Base	61.62	64.66	65.34	64.55	66.24	64.69	59.23	64.21	65.85	64.04	
Base+F	63.80	66.24	67.20	66.70	68.06	66.43	60.83	66.20	68.08	65.95	(↑ 3.0%)
Base+F+C	65.81	68.43	69.23	68.94	70.24	68.89	63.93	68.74	70.64	68.31	(↑ 6.7%)
Base+F+S	69.35	72.07	73.55	70.52	72.11	71.38	64.32	70.21	73.24	70.74	(↑ 10.5%)
Base+F+I	63.65	68.04	69.48	68.28	70.15	69.37	63.12	68.52	70.48	67.90	(↑ 6.1%)
Base+F+C+I	65.69	70.04	71.30	70.68	72.20	71.33	65.67	71.02	73.03	70.10	(↑ 9.5%)
Base+F+C+I+S	**72.24**	**75.37**	**76.26**	**74.13**	**75.37**	**74.89**	**67.85**	**74.11**	**77.54**	**74.19**	(↑ 15.8%)
Edit Similarity											
Base	84.72	86.27	86.50	86.15	86.88	85.91	83.34	86.06	86.69	85.83	
Base+F	85.82	87.21	87.59	87.24	87.79	86.96	84.35	87.21	87.83	86.89	(↑ 1.6%)
Base+F+C	86.83	88.07	88.45	88.17	88.76	88.22	85.90	88.09	88.98	87.94	(↑ 2.8%)
Base+F+S	88.19	89.60	90.18	88.59	89.66	89.12	85.79	88.70	90.15	88.82	(↑ 3.8%)
Base+F+I	86.40	87.83	88.68	88.16	88.84	88.52	85.75	88.25	89.03	87.94	(↑ 2.8%)
Base+F+C+I	87.25	88.81	89.34	89.14	89.69	89.46	86.72	89.28	90.26	88.88	(↑ 3.9%)
Base+F+C+I+S	**89.39**	**90.76**	**91.29**	**90.17**	**91.11**	**90.50**	**87.33**	**90.17**	**91.81**	**90.16**	(↑ 5.4%)



(a) When to use **(b) Which to use** **(c) How to use**

Figure 4: Performance and token numbers of different contexts. "L"denotes the number of lines included in the file context.

contexts. As described in the figure, the token length increases progressively with the addition of each context type. The basic context has the shortest token length (217.8 tokens), while the combination of all context types (+F+C+I+S) results in the longest token length, nearly 1,000 tokens. Consequently, the average throughput decreases by up to 54.4%. These results suggest that while combining different contexts can enhance API suggestions, it also incurs a trade-off by negatively impacting throughput to some extent.

(2) The average improvement gain per unit of throughput decrease differs along with the context types used in each scenario. Enriching contexts introduces a consistent throughput overhead across the three scenarios we experiment with; however, the impacts vary as shown in Figure 3 (b). For instance, combining file contexts and suffix contexts (+F+S) yields the best trade-off in the "how to use" scenario, where the exact match score improves the most per unit decrease in throughput. In the "which to use" scenario, including file contexts and code comments (+F+C) achieves the best improvement-to-throughput ratio. For the most complex "when to use" scenario, we observe that involving all types of contexts (+F+C+I+S) provides the best trade-off. Moreover, we observe that in both the "how to use" and "which to use" scenarios, file contexts and import messages (+F+I) contribute the least, with ratios of only 0.13 and 0.28, respectively. This poor trade-off results from the substantial amount of tokens added by including import messages, while the improvement in these two scenarios remains limited. However, in the "when to use" scenario, this combination improves the suggestion accuracy by 48.71%, making the improvement-to-throughput ratio 25% higher than file contexts (+F) at 1.26.

The differences among the scenarios indicate that each scenario requires a distinct context selection approach to achieve the optimal trade-off between model performance and throughput.

Finding 3: Additional contexts increase the amount of tokens in the input. Compared to basic function context, incorporating all studied contexts increases the token length by 335%, which consequently decreases the average throughput by 54%. In addition, for different scenarios, the context selection approach that achieves the best trade-off between model performance and throughput is also different.

5 DISCUSSION

5.1 Analysis of Context Selection

In the results presented in Section 4.2, we construct file contexts by including ten lines of source code preceding the function of the target API. This setting is adopted for dealing with larger-sized single files, which could exceed the token capacity of current LCMs (e.g., DeepSeek-Coder's limit of 8,192 tokens). Previous research [7, 17] has shown that additional context before the suggestion position can provide more information for model prediction. In this section, we conduct experiments to compare the effects of simply involving more file contexts with our context selection approaches, i.e., selecting contexts with various types.

Specifically, besides the setting used in Section 4.2 that file context (+F) involves ten lines of code preceding the local function (i.e., basic function context), we further evaluate LCMs with extended file contexts of 20, 40, 60, 80, and 100 lines. These settings are chosen due to their comparable context token length to our context selection approaches i.e., the token length varies from around 300 to 950, ensuring that the influence of token length on model performance is eliminated in the comparison. The comparison of these settings is presented in Figure 4.

As depicted in the figure, extending file contexts enhances API suggestion performance across all three scenarios. Specifically, when using 100 lines of code to serve as file contexts (averaging about 960 tokens in the input prompt), the exact match scores improve by 7.3%, 10.4%, and 39.2% in the respective scenarios compared to using only ten lines of file context. This remarkable improvement demonstrates that LCMs benefit from extended file contexts in API suggestion tasks. However, we can observe that as the number of tokens increases, the performance improvement becomes slower. This indicates that the benefit of simply introducing longer text without selection is limited.

Despite the obvious improvements achieved by extending file contexts, our context selection approach yields better suggestion accuracy while using tokens with similar lengths. As illustrated in the figure, the context selection curve is generally positioned in the upper left of the file-level contexts curve. Despite a context selection (+F+I) performing worse than file contexts with a similar amount (about 600) of tokens in "how to use" and "which to use" scenarios, we still have other selection choices to achieve better performance with fewer tokens in the input (e.g., +F+S). These results indicate that a carefully curated selection of contexts can outperform the simple extension of file-level contexts with the same or even smaller token length.

Therefore, we conclude that increasing the range of file contexts also benefits the API suggestion task. However, selecting contexts from various types proves to be a more efficient and effective strategy for the API suggestion task, i.e., achieving better performance with the same or even fewer input tokens.

5.2 Case Study

In this section, we conduct a case study to further illustrate the improvement brought by various contexts. The example case is presented in Figure 5. From the figure, we observe that CodeLlama 7B predicts the incorrect second argument as "*targetEntries.size()*"

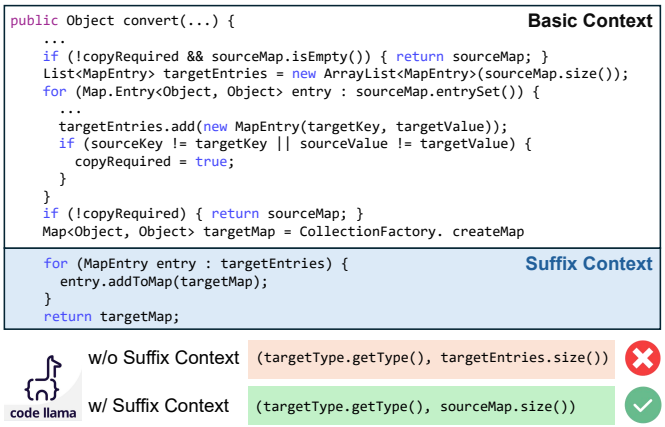


Figure 5: Case study in the "how to use" scenario, where the experimented LCM is CodeLlama 7B.

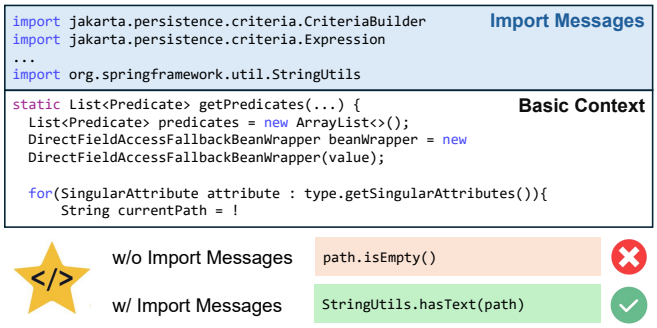


Figure 6: Case study in the "when to use" scenario, where the experimented LCM is StarCoder 15B.

when only provided with the basic function context. We hypothesize that because the newly defined object is named "*targetMap*", the model mistakenly associates its size with "*targetEntries*". However, after incorporating the suffix context, the model successfully suggests the correct usage of the API. We attribute this improvement to the loop in the suffix context, which indicates that all entries from "*sourceMap*" eventually populate "*targetMap*". This helps the model recognize that "*targetEntries*" is not the source of the initial map size.

We also provide a case study in the "when to use" scenario, where the example is presented in Figure 6. From the figure, we find that with basic context, StarCoder 15B defaults to a common method available on the "*path*" object and calls "*isEmpty()*", a widely known method to check if a string is empty. Contrarily, involving import messages facilitates the model in successfully predicting the desired API calls. We suppose that, after incorporating import messages, the model recognizes that "*StringUtils*" provides utility methods for string operations, and then uses the "*hasText*" API.

5.3 Implications of Findings

5.3.1 Implications for Researchers. Our research demonstrates that current LCMs perform variously in the three scenarios of the API suggestion task. With well-designed context selection, the model performance can be substantially improved. However, as shown in

RQ1 and RQ2, there exists a remarkable gap between the performance in the "when to use" and the other scenarios, indicating the need for further research and improvement in this direction.

Our results also reveal the potential research directions in the era of LCM for the community. Specifically:

- **Exploring more effective context selection approaches**. This paper concentrates on selecting contexts within the code file. As reported in the work [27, 34], cross-file information can potentially enhance API suggestions. Therefore, it is essential to explore more effective context selection approaches that reduce token length while simultaneously improving model performance. Future research should investigate methods such as analyzing file dependencies and selecting context from other related files within the project.
- **Paying more attention to the "when to use" scenarios**. Previous research has predominantly concentrated on suggesting which APIs to use, ignoring other scenarios and challenges developers encounter when working with APIs. Our results in RQ1 and RQ2 demonstrate that LCMs exhibit varying performance across different scenarios. Notably, in the "when to use" scenario, we can find that the average exact match metric is 16% and 22.4% lower than that in the "which to use" and "how to use" scenarios (with the optimal context selection), respectively. Therefore, these findings suggest that researchers should broaden their focus to include a wider range of scenarios in API suggestion tasks, better improving productivity for developers.

5.3.2 Implications for Developers. In this section, we take both performance and efficiency into account and provide insights on model selection and context selection.

Implications on model selection. In Section 4.2, we establish that larger models generally outperform smaller ones and report the average throughput of nine LCMs with the provided contexts. To further inform model selection, we present the throughput of LCMs that are grouped according to their sizes in Figure 7 (the detailed results of individual LCMs are involved in our anonymous repository). The figure reveals that all experimented LCMs exhibit similar trends across different context combinations. Notably, we observe that smaller LCMs (e.g., StarCoder 3B and DeepSeek-Coder 1.3B) maintain higher throughput across all contexts (+F+C+I+S) compared to larger LCMs (e.g., StarCoder 15B and DeepSeek-Coder 33B). These findings suggest that if computational resources cannot accommodate large LCMs with over 10 billion parameters, opting for smaller models with enriched contexts is a more efficient and effective choice in the API suggestion task.

In addition, the combination of different LCMs according to scenarios is also worth exploring. As demonstrated in Section 4.2, in simpler scenarios such as the "how to use" scenario, the performance gap between large models and small models is relatively small. Therefore, for less demanding scenarios such as "how to use", smaller models can be utilized to achieve satisfactory performance while conserving computational resources and reducing latency. Conversely, for more complex scenarios that require higher accuracy and nuanced understanding, larger models can be employed to leverage their superior capabilities. This approach allows for efficient resource management and optimized performance across a variety of use cases.

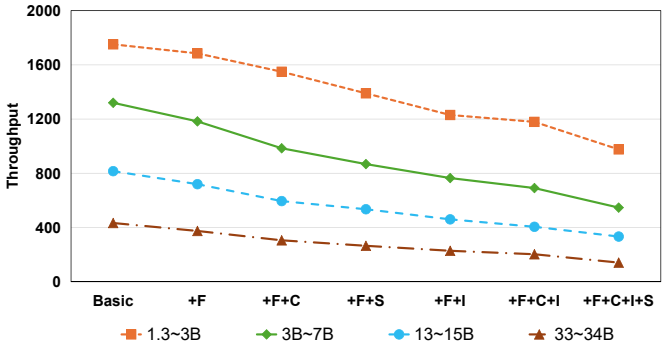


Figure 7: Illustration of throughput of the different contexts, in which the models are grouped by their sizes for clarity.

Implications on context selection. According to Figure 3, we observe that different contexts contribute variously across different scenarios, indicating that there is no universal solution for context selection in API suggestion tasks. For instance, combining all available contexts (+F+C+I+S) yields the best performance in the three scenarios. However, this improvement is less satisfactory when considering the 54% throughput overhead in the "how to use" scenario.

Therefore, when deploying API suggestion services with limited computational resources that cannot accommodate all studied contexts, developers could tailor context selection to specific scenarios. If the latency associated with loading all contexts (+F+C+I+S) is unacceptable, using comments (+C) and suffix contexts (+S) can be effective choices for the "how to use" and "which to use" scenarios. For the "when to use" scenario, combining comments and import messages (+C+I) offers a good trade-off between performance and computational efficiency.

5.4 Discussion about Evaluation Metrics

In this paper, we use edit similarity to evaluate the string-level similarity of LCMs' outputs and ground truth API calls. Specifically, we use the metric due to the following reasons:

- API suggestion is a specific scenario in code completion tasks and edit similarity is a popular metric used in existing research [23].
- The edit similarity metric is meaningful in API suggestion tasks. For instance, LCMs may predict a wrong but similar API to use, e.g., the target is getField but predicting getFields. In this situation, the prediction has a high edit similarity, indicating that users just need to delete an "s" after accepting LCMs' output. Therefore, edit similarity can help identify a "plausible" answer that is close to the users' needs.
- The ranking metric is also helpful; however, calculating this metric requires LCMs to generate multiple answers, which brings substantial overhead on time consumption. We will involve this metric in future work.

5.5 Threats to Validity

We have identified the following major threats to validity:

Limited LCMs. The experiments in this paper are based on open-source popular LCMs, which may bring bias in the results. To mitigate this issue, we select nine LCMs with various model sizes to control the threat. Furthermore, the improvement of context selection is model-agnostic, making our findings easily generalize to other LCMs.

Limited API source. In this paper, we focus on the Spring Framework in Java to evaluate the API suggestion task. This choice is grounded that SpringFramework is one of the most widely used frameworks in the Java ecosystem, offering a comprehensive set of features for building robust and scalable applications. In addition, the diversity and complexity of SpringFramework APIs provide a challenging testbed for evaluating the performance of large code models (LCMs) in API suggestion tasks.

Potential data leakage. In this paper, the data used for training LCMs are not publicly available, so that we can hardly determine whether there exists data leakage in these models. However, our experiments reveal that merely providing function contexts to LCMs cannot yield promising performance. Therefore, we believe that the results produced by LCMs in the benchmark are not from simply memorizing data.

6 RELATED WORK

6.1 API Suggestion

Application Programming Interfaces (APIs) are crucial for enabling developers to integrate existing functionalities rather than building them from scratch. Various automated API method recommendation techniques have been developed to assist developers in writing correct APIs [26, 31, 44]. McMillan et al. [26] propose portfolio, an API recommendation tool that aids programmers in locating relevant code snippets that fulfill high-level requirements specified in a query. Gu et al. [15] propose DeepAPI, which utilizes a deep learning model to suggest API usage sequences for a given natural language query. CLEAR [40] uses contrastive learning and BERT sentence embedding similarity to first identify a set of candidate Stack Overflow posts and then re-ranks to recommend the top-N APIs. MEGA [5] employs heterogeneous graphs to learn the matching scores between methods and APIs for recommending related APIs. Different from previous works that mainly focus on API recommendation, our work encompasses more scenarios in API suggestion including when, which, and how to use.

6.2 Large Code Models

Recently, the emergence of Large code models has revolutionized various software engineering tasks [9, 12, 33, 41]. StarCoder is a foundation code model [22] trained on the mixture of source code and natural language texts. Its training data incorporate more than 80 different programming languages as well as text extracted from GitHub issues and commits and from notebooks. Code Llama [33] is a foundation model developed by Meta that helps generate and understand programming code. It builds on the capabilities of the original LLaMA models and extends the context length to 16K. DeepSeek Coder [16] has a range of open-source LCMs with sizes varying from 1.3B to 33B. It is trained from scratch on a curated code corpus with 2T tokens. Apart from the foundation models, various fine-tuning [24, 39, 41] and prompting techniques [3, 13] are also proposed to make full use of LCMs for software engineering tasks. For example, Magicoder [41] proposes to synthesize instruction data from open-source code snippets for effectively tuning large code models for code generation. Ahmed et al. [3] propose to augment the prompt with repository information and data flow for boosting the performance of code summarization. TypeGEN [30] uses static analysis results and chain-of-thought prompts to guide LLMs in type inference. ChatUniTest [42] extracts essential information and creates an adaptive focal context for LCMs to generate test cases.

6.3 LLMs in API Suggestion

Huang et al. [18] propose to combine knowledge graphs and LLMs to find APIs based on natural language queries. CAPIR [25] adopts a "divide-and-conquer" strategy to recommend APIs for coarse-grained developmental requirements. APIGen [6] is a generative method that utilizes improved in-context learning to directly generate the API name. Recently, Nashid et al. [27] study the performance of LLMs in generating APIs for unseen repositories. Different from previous works that only focus on API recommendation (i.e., which to use), our work presents a more comprehensive study of different API usage scenarios in real-world software development.

7 CONCLUSION

In this paper, we have conducted a systematic evaluation of LCMs in API suggestion, proposing three distinct scenarios, when, which, and how to use an API, to comprehensively assess their capabilities. Our experiments on a diverse benchmark dataset have revealed that LCMs perform best in the "how to use" scenario and benefit substantially from enriched context, albeit at the cost of increased token length and reduced throughput. Our findings offer valuable insights into the strengths and limitations of LCMs in API suggestion and highlight the critical role of context in enhancing their performance.

ACKNOWLEDGMENT

This research is supported by the National Natural Science Foundation of China under project (No. 62472126), Natural Science Foundation of Guangdong Province (Project No. 2023A1515011959), Shenzhen-Hong Kong Jointly Funded Project (Category A, No. SGDX20230116091246007), Shenzhen Basic Research (General Project No. JCYJ20220531095214031), Shenzhen International Science and Technology Cooperation Project (No. GJHZ20220913143008015). This research is also supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14206921 of the General Research Fund).

REFERENCES

[1] [n. d.]. *MVN Repositories*. https://mvnrepository.com/open-source/web-frameworks.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. *arXiv preprint arXiv:2403.02310* (2024).

[3] Toufique Ahmed, Kunal Suresh Pai, Premkumar T. Devanbu, and Earl T. Barr. 2024. Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization). In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 220:1–220:13.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. (2020).

[5] Yujia Chen, Cuiyun Gao, Xiaoxue Ren, Yun Peng, Xin Xia, and Michael R. Lyu. 2023. API Usage Recommendation Via Multi-View Heterogeneous Graph Representation Learning. *IEEE Trans. Software Eng.* 49, 5 (2023), 3289–3304.

[6] Yujia Chen, Cuiyun Gao, Muyijie Zhu, Qing Liao, Yong Wang, and Guoai Xu. 2024. APIGen: Generative API Method Recommendation. *arXiv preprint arXiv:2401.15843* (2024).

[7] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An empirical study on the usage of bert models for code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 108–119.

[8] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.

[9] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533* (2023).

[10] Jaroslav Fowkes and Charles Sutton. 2016. Parameter-free probabilistic API mining across GitHub. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 254–265.

[11] Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Lunyiu Nie, Xin Xia, and Michael R. Lyu. 2023. Code Structure-Guided Transformer for Source Code Summarization. *ACM Trans. Softw. Eng. Methodol.* 32, 1 (2023), 23:1–23:32.

[12] Shuzheng Gao, Wenxin Mao, Cuiyun Gao, Li Li, Xing Hu, Xin Xia, and Michael R Lyu. 2024. Learning in the wild: Towards leveraging unlabeled data for effectively tuning pre-trained code models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[13] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R Lyu. 2023. What makes good in-context demonstrations for code intelligence tasks with llms?. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 761–773.

[14] Golara Garousi, Vahid Garousi-Yusifoğlu, Guenther Ruhe, Junji Zhi, Mahmoud Moussavi, and Brian Smith. 2015. Usage and usefulness of technical software documentation: An industrial case study. *Information and software technology* 57 (2015), 664–682.

[15] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. ACM, 631–642.

[16] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence. *CoRR* abs/2401.14196 (2024).

[17] Xincheng He, Lei Xu, Xiangyu Zhang, Rui Hao, Yang Feng, and Baowen Xu. 2021. Pyart: Python api recommendation in real-time. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1634–1645.

[18] Qing Huang, Zhenyu Wan, Zhenchang Xing, Changjing Wang, Jieshan Chen, Xiwei Xu, and Qinghua Lu. 2023. Let's Chat to Find the APIs: Connecting Human, LLM and Knowledge Graph through AI Chain. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 471–483.

[19] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[20] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *CoRR* abs/2305.06161 (2023).

[21] Fang Liu, Zhiyi Fu, Ge Li, Zhi Jin, Hui Liu, Yiyang Hao, and Li Zhang. 2024. Non-Autoregressive Line-Level Code Completion. *ACM Transactions on Software Engineering and Methodology* (2024).

[22] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian J. McAuley, Han Hu, Torsten Scholak, Sébastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, and et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. *CoRR* abs/2402.19173 (2024).

[23] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. (2021).

[24] 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).

[25] Zexiong Ma, Shengnan An, Bing Xie, and Zeqi Lin. 2024. Compositional API Recommendation for Library-Oriented Code Generation. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. 87–98.

[26] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. ACM, 111–120.

[27] Noor Nashid, Taha Shabani, Parsa Alian, and Ali Mesbah. 2024. Contextual API Completion for Unseen Repositories Using LLMs. *arXiv preprint arXiv:2405.04600* (2024).

[28] Konstantinos Papaioannou and Thaleia Dimitra Doudali. 2024. The Importance of Workload Choice in Evaluating LLM Inference Systems. In *Proceedings of the 4th Workshop on Machine Learning and Systems*. 39–46.

[29] Yun Peng, Shuqing Li, Wenwei Gu, Yichen Li, Wenxuan Wang, Cuiyun Gao, and Michael R Lyu. 2022. Revisiting, benchmarking and exploring api recommendation: How far are we? *IEEE Transactions on Software Engineering* 49, 4 (2022), 1876–1897.

[30] Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R. Lyu. 2023. Generative Type Inference for Python. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 988–999.

[31] Mohammad Masudur Rahman, Chanchal K Roy, and David Lo. 2016. Rack: Automatic api recommendation using crowdsourced knowledge. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 349–359.

[32] Vipula Rawte, Amit Sheth, and Amitava Das. 2023. A survey of hallucination in large foundation models. *arXiv preprint arXiv:2309.05922* (2023).

[33] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023).

[34] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*. PMLR, 31693–31715.

[35] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 1433–1443.

[36] Ze Tang, Jidong Ge, Shangqing Liu, Tingwei Zhu, Tongtong Xu, Liguo Huang, and Bin Luo. 2023. Domain Adaptive Code Completion via Language Models and Decoupled Domain Databases. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 421–433.

[37] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva,

Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023).

[38] treesitter. 2024. treesitter. https://github.com/tree-sitter/tree-sitter.

[39] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R Lyu. 2022. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering*. 382–394.

[40] Moshi Wei, Nima Shiri Harzevili, Yuchao Huang, Junjie Wang, and Song Wang. 2022. Clear: contrastive learning for api recommendation. In *Proceedings of the 44th International Conference on Software Engineering*. 376–387.

[41] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source Code Is All You Need. *CoRR* abs/2312.02120 (2023).

[42] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. ChatUniTest: a ChatGPT-based automated unit test generation tool. *CoRR* abs/2305.04764 (2023).

[43] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2023. WizardLM: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations*.

[44] Yu Zhou, Xinying Yang, Taolue Chen, Zhiqiu Huang, Xiaoxing Ma, and Harald Gall. 2021. Boosting API recommendation with implicit feedback. *IEEE Transactions on Software Engineering* 48, 6 (2021), 2157–2172.