Wuhan University Journal of Natural Sciences

2026, Vol.31 No.1, 091-100

Article ID 1007-1202(2026)01-0091-10 DOI https://doi.org/10.1051/wujns/2026311091

Cite this article: LIU Zhiping. On-Demand API Non-Human-Reliant Tutorial Generation by LLM-Based Across-Language Knowledge Transfer[J]. *Wuhan Univ J of Nat Sci*, 2026, 31(1): 91-100.

On-Demand API Non-Human-Reliant Tutorial Generation by LLM-Based Across-Language Knowledge Transfer

□ **LIU Zhiping**

College of Information Engineering, Gandong University, Fuzhou 344000, Jiangxi, China

Abstract: API (Application Programming Interface) documentation often only describes individual APIs and lacks information on complex API relations and code examples. Retrieval-based and generation-based methods can both produce documentation that includes API relationship descriptions and code examples. However, they are limited by the richness of available API resources. As a result, they struggle to be effective when dealing with resource-scarce languages such as Kotlin. We propose an on-demand API tutorial generation method for resource-scarce languages, transferring API knowledge from a resource-rich language like Java to Kotlin using an AI chain. Evaluating our method on 500 Kotlin APIs, we generated more API documents than the state-of-the-art retrieval-based method ADECK and the generate-based method gDoc. The number of API guidelines generated by our method is 37 times that of ADECK and 1.6 times that of gDoc. Compared with the scheme that did not adopt the knowledge transfer strategy, the success rate of our method has increased by 31.25 percentage points. This demonstrates the feasibility and potential of using LLMs to create new API knowledge across languages.

Key words: on-demand; API tutorial; API relation; large language model (LLM)

CLC number: TP311.52; TP391.2

0 Introduction

API (Application Programming Interface) encapsulate specific implementations of functions and provide standardized methods of invocation, allowing developers to directly call the API to achieve specific functionalities without having to write redundant logic themselves. To effectively utilize an API, developers usually refer to the API documentation to understand the functions and usage of the API. However, the existing API documentation only provides code examples and functional descriptions for each API separately, without explaining the complex relationships among multiple APIs[1], this results in the existing API documentation being difficult to be effective in practical applications, because most tasks cannot be fully accomplished by a single API alone in practical applications, developers often need a deep understanding of the complex relationships between APIs. For example, developers usually need to understand the collaboration relationships between multiple APIs in order to integrate multiple APIs to complete tasks together[2-3], and the same is true in code optimization tasks[4-5]. They also need to un-

Received date: 2025-03-16

Foundation item: Supported by the High-Level Research Fund (12225000404)

Biography: LIU Zhiping, female, Professor, research direction: software engineering. E-mail: zhipingliu777@163.com

derstand the subtle differences between multiple APIs (such as input parameters) in order to better choose the more appropriate APIs in specific scenarios.

To obtain more documents of related APIs with API relation descriptions and code examples, in 2017, Robillard *et al*[6] proposed on-demand developer documentation (OD3), aims to utilize information retrieval and knowledge extraction technologies to automatically assemble scattered API knowledge into an on-demand API document that meets diverse API knowledge requirements. Inspired by OD3, Zhang *et al*[7] proposed ADECK. It is a retrieval-based method that uses custom heuristic rules to retrieve, extract and filter relevant API knowledge from online resources such as Stack Overflow, API documentation and tutorials, and forms structured API documentation through the integration of API knowledge. Different from ADECK, Wang *et al*[8] proposed gDoc, which leverages a large corpus of API documentation to train a Seq2Seq-based translation model capable of generating structured API documentation.

Some languages, such as Java, have accumulated rich resources regarding complex API relationships (for example, Stack Overflow). However, languages like Kotlin, which have a relatively short history and lower popularity, often lack sufficient API resources, making it difficult for existing approaches to generate effective API documentation. Retrieval-based methods such as ADECK rely heavily on the availability of reliable and diverse online materials. When such resources are limited, these methods typically encounter retrieval errors or return irrelevant results. For instance, when searching for information about API *kotlin. text. partition*(), information about *kotlin. collections. partition*() is retrieved (the information is shown in Fig. 1). Although these two APIs share the same function name *partition*(), they operate on different types of data. The *kotlin. text. partition* () is used for splitting a string based on a condition applied to its characters (the specific description of *kotlin. text.partition*() is shown in Fig. 1). The *kotlin.collections. partition*() is used for dividing a collection into two groups based on a condition applied to its elements. Generation-based methods such as gDoc rely on large-scale, high-quality API documentation to train their models. However, in resource-scarce languages like Kotlin, the lack of sufficient training data[9-11] significantly undermines the model ' s ability to understand API semantics[12-13]. Therefore the LLM (Large Language Model) may lack sufficient knowledge to generate API tutorials. For example, directly generating an API tutorial with LLM for *kotlin.text.partition*() may generate an incorrect output like *kotlin.collection.partition*(). The functions of these two APIs are similar and there is no functional collaboration relationship between them.

Programming languages often share similarities due to inheritance, such as Kotlin's object-oriented style and API design similar to Java, which enhances their interoperability. Inspired by these similarities, we propose a knowledge transfer method to generate on-demand API tutorials for resource-scarce languages. Our approach generates on-demand API tutorials for resource-scarce languages by leveraging the API knowledge of resource-rich languages (e.g., Java) and transferring it to resource-scarce languages (e.g., Kotlin) through an AI chain. For example, our method generates an on-demand API tutorial with an LLM for *kotlin. text. partition*(), including a code example and a API relation description that reflects the function-collaboration between *kotlin.text.partition*() and *kotlin. text. groupingBy*(). The *partition*() divides characters into those that meet a condition and those that don't, and it is often used for initial filtering of data sub-

kotlin.text.partition()	**Splits** the **original string/char sequence** into **a pair of strings/char sequences**, where *first* string/char sequence contains characters for which predicate yielded true, while *second* string/char sequence contains characters for which predicate yielded false.
kotlin.collection.partition()	**Splits** the **original collection** into **a pair of lists**, where *first* list contains elements for which predicate yielded true, while *second* list contains elements for which predicate yielded false.
kotlin.collection.groupingBy()	**Creates a Grouping source** from a **char sequence** to be used later with one of group-and-fold operations using the specified keySelector function to extract a key from each character.

Fig. 1 Function description of Kotlin API

sets. On the other hand, the *groupingBy*() further groups and aggregates these subsets based on specified keys.

The main contributions of this paper are as follows:

• Shift of focus: Unlike previous approaches that address issues through API usage individually, our work emphasizes the importance of the on-demand API tutorial for an API of interest and its related APIs with code examples, which closely match developers' needs in real-world applications.

• Generation-based method: In contrast to retrieval-based methods, our approach adopts a generation-based method, which overcomes the limitations imposed by scarce resources.

• Knowledge transfer: We propose knowledge transfer method to bridge the gap between resource-scarce languages and LLMs.

Our research is supported by a series of experiments, providing robust evidence of the innovation and effectiveness of our method in generating on-demand API tutorials.

1 Method

This section details our approach based on knowledge transfer, covering hierarchical task breakdown and hierarchical module decomposition, prompt design for AI units, and a running example.

1.1 Hierarchical Task Decomposition

Generating on-demand API tutorials is a complex process that requires decomposing the root module into multiple sub-modules. As shown in Fig. 2, on-demand API tutorial generation is divided into two sub-modules: the API generator (the workflow is shown in Fig. 3) and the API tutorial generator (the workflow is shown in Fig. 4). The API generator produces a related API within the same language from a given API. The API tutorial generator then creates text and code to reflect the relationship between these APIs, forming a complete API document. This implementation specifically targets Kotlin and covers three types of API relations: behavior difference, function-collaboration, and logic-constraint. The definitions of these relations can be found in previous works[1,14].

1.2 Hierarchical Module Decomposition

To ensure the control ability of our approach, we follow the single responsibility principle by decomposing sub-modules into independent units, categorized into AI units and Non-AI units. The AI unit leverage the capabilities of Large Language Models (LLMs) by manual prompt to achieve certain functions, and the Non-AI unit are implemented through hard coding to achieve. Next, we detail the functional units in the API Generator and API Tutorial Generator sub-modules.

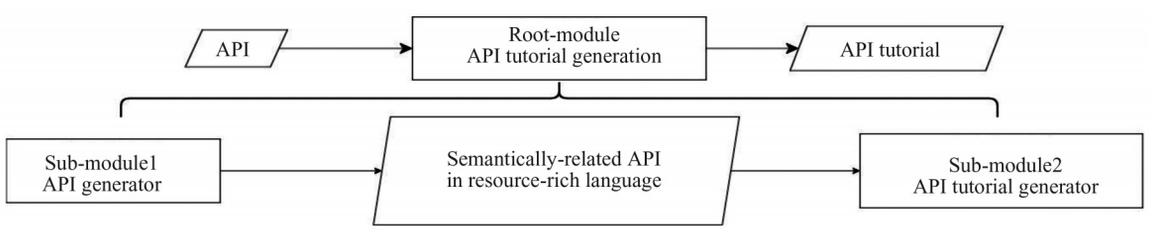

Fig. 2 Overall framework of our approach

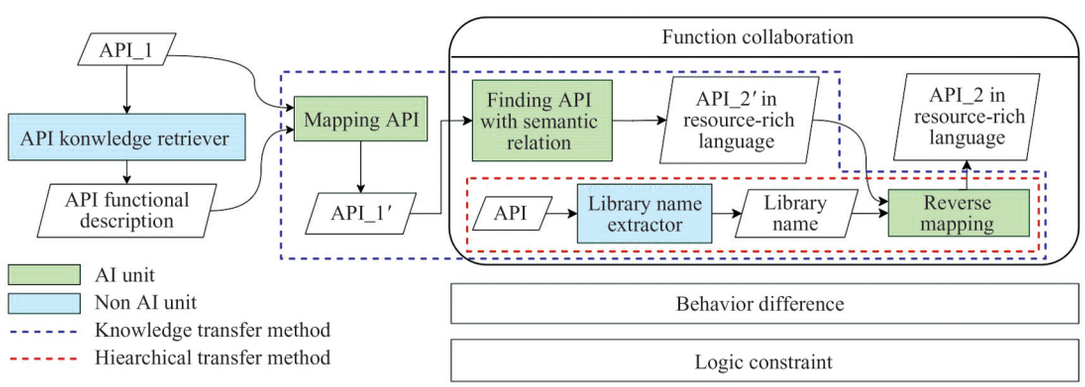

Fig. 3 The workflow of API generator

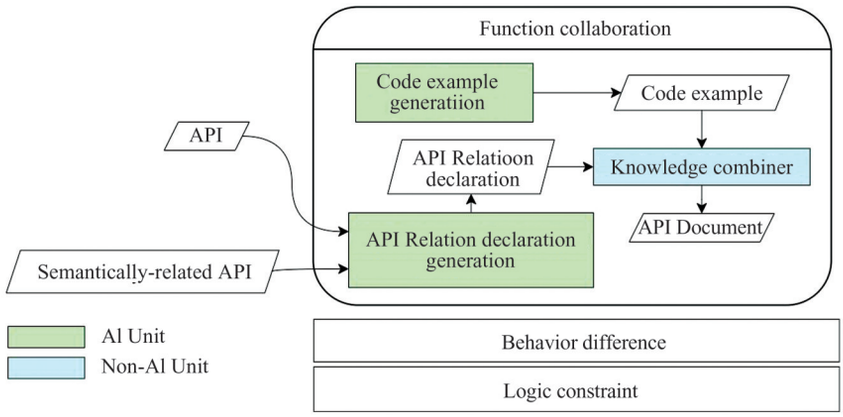

Fig. 4 The workflow of API tutorial generator

1.2.1 API generator

The API generator sub-module consists of three AI units (Mapping API, Finding API with Semantic Relation, and Reverse Mapping) and two Non-AI units (API Knowledge Retriever and Library Name Extractor). Next is an introduction to each unit in API Generator:

• API Knowledge Retriever. This unit retrieves the functional description of the input API_1 from Kotlin knowledge (API name and corresponding function description) we crawl in Kotlin documentation.

• Mapping API. Based on API_1 and its functional description, this unit aims to find a functionally-similar API_1' in the resource-rich language (e. g., Java). Functionally-similar relation is defined that two API entities have similar usage.

• Finding API with Semantic Relation. This unit retrieves API_2', which has a specific semantic relation (behavior difference, function-collaboration, and logic-constraint.) with API_1', from the resource-rich language. Behavior difference relation is defined that two similar API entities behave differently when completing the same task. Function Collaboration relation is defined that two API entities should be used together when accomplishing a task. Logic Constraint relation is defined that one API should be called before or after using another API.

• Library Name Extractor. This Non-AI unit aims to extract the library names associated with the API_1 of the resource-scare language. The usage of the API library name depends on the specific transfer strategy employed, which is further explained in Section 1.2.2.

• Reverse Mapping. This unit maps API_2' into a resource-scarce language API (API_2) with the same library name as API_1.

1.2.2 Hierarchical transfer strategy

Identifying API similarities between programming languages, like Java and Kotlin, can be challenging for LLMs due to the knowledge gap. This often leads to multiple results when directly mapping a single API. For instance, when mapping *java. util. stream. Stream. map*() to Kotlin using LLM, the generated results may include *kotlin.collections.map*(), *sequences.map*(), or *text.map*().

To improve API mapping accuracy, we propose a hierarchical transfer strategy for the Reverse Mapping unit. This strategy refines LLM searches by first mapping the library name (e.g., *java.util.stream.Stream*()) to Kotlin, then mapping the method (e.g., *map*()) to the preliminary result, resulting in *kotlin.sequence.map*().

The hierarchical transfer strategy is divided into three types: "hierarchical-front", applied to the Mapping API unit, "hierarchical-behind", applied to the Reverse Mapping unit, and "hierarchical-both", applied to both units simultaneously. These strategies differ in how they determine the mapping scope. The hierarchical-front transfer strategy establishes the mapping scope by first finding a similar library name in the resource-scarce language based on the the library name of input API before conducting the mapping. For instance, in the Mapping API unit, we extract the library name from the input API (e.g., *kotlin.collections.partition*()) and map it to a similar library name (*java. util. stream. collectors*) in the resource-rich language (Java). Subsequently, the input API (*kotlin.collections.partition*()) is mapped to the corresponding library name (*java.util.stream.Collectors*) in the resource-rich language, resulting in a similar API (*java. util. stream. Collectors. partitioningBy*()). On the other hand, the hierarchical-behind transfer strategy directly extracts the library name from the input API provided to the Mapping API unit. This extracted library

name is then used in the Reverse Mapping unit to perform API mapping. For example, in the Reverse Mapping unit, we extract the library name from the input (e.g., *kotlin.collections.partition*()) of the Mapping API unit. Then, in the Reverse Mapping unit, we use this library name (*kotlin.collections*) to map semantically related APIs (*java.util.stream.Collectors.groupingBy*()) to the corresponding library in the resource-scarce language (Kotlin) and find a similar API (*kotlin.collections.groupby*()) in the resource-scarce language. Overall, the hierarchical transfer strategy improves the accuracy of API mappings by narrowing the search scope and leveraging the library name of the APIs.

1.2.3 API tutorials generator

The API tutorial Generator sub-module consists of two AI units (API relation description Generation and Code Example Generation) and a Non-AI unit (Knowledge Combiner). There is a description of each unit in the API tutorials Generator:

• API relation description Generation. It generates text that reflects a relation between API_1 and API_2.

• Code Example Generation. It generates code based on the API relation text, illustrating API relation.

• Knowledge Combiner. It combines the generated text and code to form the API tutorial of the input API.

1.3 Prompt Design for AI Unit

To implement each AI unit, we use the in-context learning approach, leveraging the vast knowledge within LLMs. We use prompts that contain examples and task descriptions to enable this. Below, we detail the prompt design for each AI unit in every module.

1.3.1 AI unit in API generator module

• Prompt Design for Mapping API. This AI unit aims to map APIs in a resource-scarce language to similar ones in a resource-rich language that the LLM is familiar with. As shown in Fig. 5, the prompt of the Mapping API unit includes a task description of and five input-output examples. The prompt takes an API (e.g., *kotlin.sequences.find*()) and its description as input (yellow part) and generates an API (pink part) that is functionally-similar to the input API.

• Prompt Design for Finding API with Semantic Relation. In Fig. 6, the Finding API with Function Collaboration Relation unit generates a resource-rich language API that has a specific relation with the input API from resource-rich language. Its prompt includes a task description and five task input-output examples. The input (yellow part) to prompt is a Java API, and the output is an API (pink part) that has a certain API relation with the input API.

• Prompt Design for Reverse Mapping. This unit aims to map APIs in resource-rich language back to the resource-scarce one. As shown in Fig. 7, the prompt of the Reverse Mapping unit includes a task description and five input-output examples. The input (yellow part) includes a library name of Kotlin API and a Java API, and the output is a Kotlin API (pink part) that has the same Kotlin library name and is functionally similar to the input API.

1.3.2 AI unit in API tutorials generation module

• Prompt Design for API relation description Generation. This AI unit generates API relation descriptions

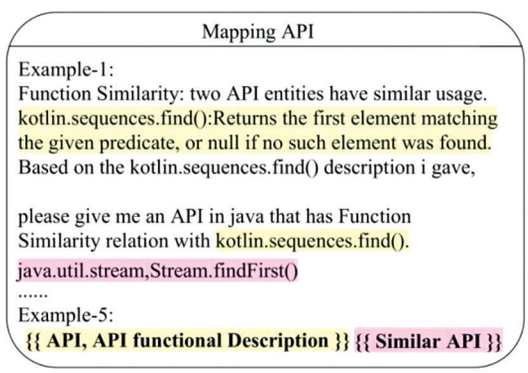

Fig. 5 Mapping API unit

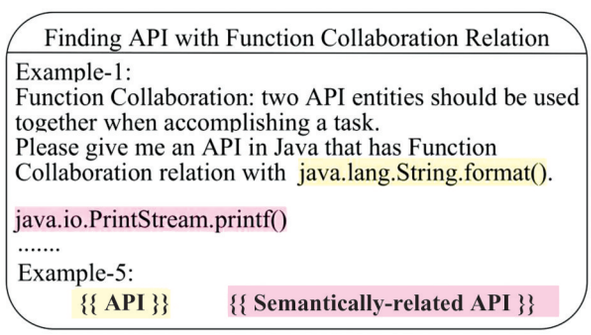

Fig. 6 Finding API with semantic relation unit
(in resource-rich language)

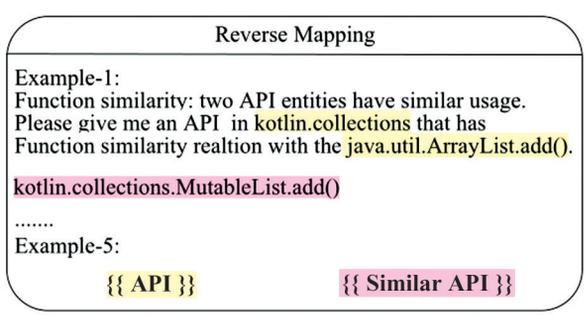

Fig. 7 Reverse mapping unit
(resource-rich to resource-scarce language)

(pink) between two APIs in a resource-scarce language (e.g., Kotlin). As shown in Fig. 8, the prompt includes a task description and output format requirements. For instance, when generating function collaboration relation tutorials, the prompt contains specific declarations (blue) and requirements (green).

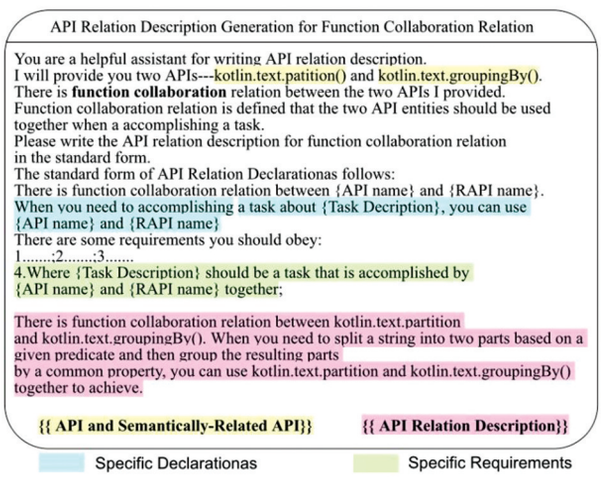

Fig. 8 API relation description generation unit

• Prompt Design for Code Example Generation. This unit is designed to guide the LLM in generating a code example (pink part) based on the given API text (yellow part). As shown in Fig. 9, the prompt of Code Example Generation for Function Collaboration Relation unit includes a task description and some output format requirements.

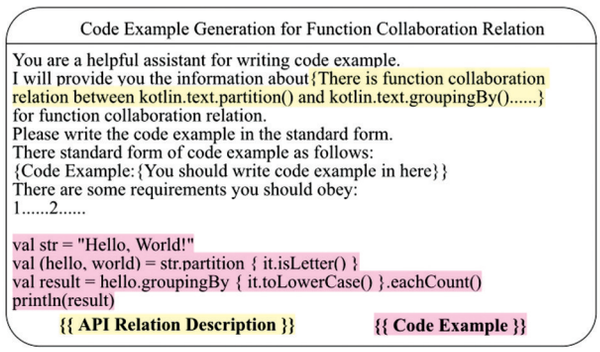

Fig.9 Code example generation unit

1.4 Running Example

Figure 10 illustrates our approach with an example. First, the API Knowledge Retriever retrieves information about *kotlin.text.partition*(). The Mapping API unit then finds a Java API *java.util.stream.Collectors.partitioningBy*() that similar to input API. The Finding API with Semantic Relation unit find *java.util.stream.Collectors.groupingBy*() as having a function-collaboration relation. The Library Name Extractor extracts *kotlin.text* from the Kotlin API. Using the Reverse Mapping unit, *groupingBy*() method is mapped back to *kotlin.text.groupingBy*(). The API Relation Description Generation unit produces a description of the function-collaboration relation, which the Code Example Generation unit uses to create a code example. Finally, the Knowledge Combiner unit integrates the API relation text and code example to generate the API tutorial.

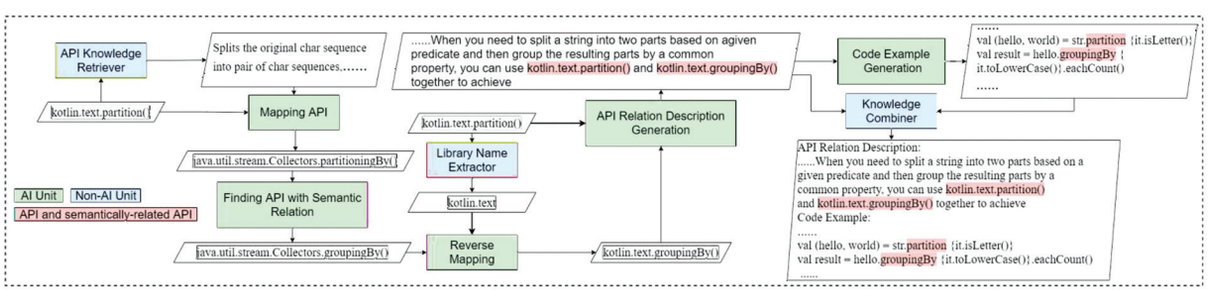

Fig.10 Running example

2 Experiment Setup

2.1 Research Questions

We set up three questions to verify the performance of our approach for generating the on-demand API tutorial for an API of interest and its related APIs with API relation descriptions and code examples.

(1) How effective is the API tutorial generation of our approach;

(2) How effective are the strategies in our approach;

(3) How different large language models differ in our approach.

2.2 Data Preparation

We selected Kotlin as the target language due to its

limited online API resources since its 2011 release. Java, with its extensive resources and similar syntax due to inheritance, serves as the source language. We created Dataset-1 by extracting 500 APIs and their descriptions from Kotlin documentation and then randomly selected 80 entries to form Dataset-2.

2.3 Baseline

In order to validate the effectiveness of our approach in API Tutorial Generation, we compare it with the state-of-art method, ADECK[7] and gDoc[8]. ADECK, retrieves extracts, and filters relevant documents from Stack Overflow containing API relation declarations and code examples related to input Java APIs, supplementing the API tutorial. The gDoc proposes a Seq2Seq model-based translation strategy to automatically generate structured API documentation.

In order to verify the effectiveness of the strategy (knowledge transfer strategy and hierarchical transfer strategy) proposed in our method to improve the API document generation effect, we designed two different variations: Non-KT-based method and direct transfer-based method. Non-KT-based method does not depend on knowledge transfer (Non-KT). Instead, it directly utilizes the LLM (GPT-3.5) for the purpose of creating on-demand API tutorials for resource-scarce programming languages. Direct transfer-based method does not involve this logical step and directly generates the API when mapping API between resource-scarce languages and resource-rich languages.

In order to evaluate the impact of different large language models on API documentation generation, we replace the GPT-3.5 model used in our approach with two alternative LLMs: ChatGPT (https://ChatGPT.com) and GPT4[15].

2.4 Metric

In this paper, we use success rate as the evaluation metric for all research questions. The success rate is calculated by dividing the number of correctly generated APIs (or text/code) by the total number generated. Human annotations determine correctness. For related APIs, annotators assess if the API exists and has a genuine semantic relationship (Function Similarity, Function Collaboration, Behavior Difference, or Logic Constraint) with the input API. For text, annotators check if it accurately reflects the API relationship. For code, correctness depends on whether the examples compile accurately and correspond to the input and found API.

3 Experiment Result

3.1 The API Tutorial Generation

To determine the effectiveness of our approach to generate API tutorials, we conducted an experiment to assess the the quantity and quality of API documents generated. In this experiment, we conducted a comparison of our proposed method against the ADECK and the gDoc using Dataset-1 to assess the number of supplemented API documents generated by each method. To ensure accurate evaluations, we enlist the expertise of eight Ph D students with over 3 years of experience in Java development to annotate the results.

Table 1 shows the quantity and success rate of supplemented API documents for Dataset-1 using ADECK, gDoc and our method. ADECK, a state-of-the-art retrieval-based technology, provides only 10 API documents with related APIs, relation descriptions, and code examples, highlighting its limitations due to reliance on available online resources. When such resources are limited, ADECK typically encounter retrieval errors or return irrelevant results. For instance, when searching for information about API *kotlin.text.partition*(), information about *kotlin.collections.partition*() is retrieved, as shown in Fig. 11. The gDoc, a state-of-the-art generate-based technology, provides 236 API documents with related APIs, relation descriptions, and code examples. When the large language model lacks sufficient training data, its ability to understand the API semantics will be greatly reduced, thereby leading to incorrect inference of the API. For instance, directly generating an API tutorial with LLM for *kotlin.text.partition*() may generate an incorrect output like *kotlin.collection.partition*(). In contrast, our method generates 378 API documents, which is 37 times that of ADECK and 1.6 times that of gDoc, and the success rate reaches 75.6%. Our method successfully generates on-demand API tutorials for resource-scarce languages by using the API knowledge of resource-rich languages (e.g., Java) and transferring it to resource-scarce languages (e. g., Kotlin)

Table 1 Success rate in different approaches

Method	Number of generated API tutorials	Success rate/%
ADECK	10	1.22
gDoc	236	47.2
Ours	378	75.6

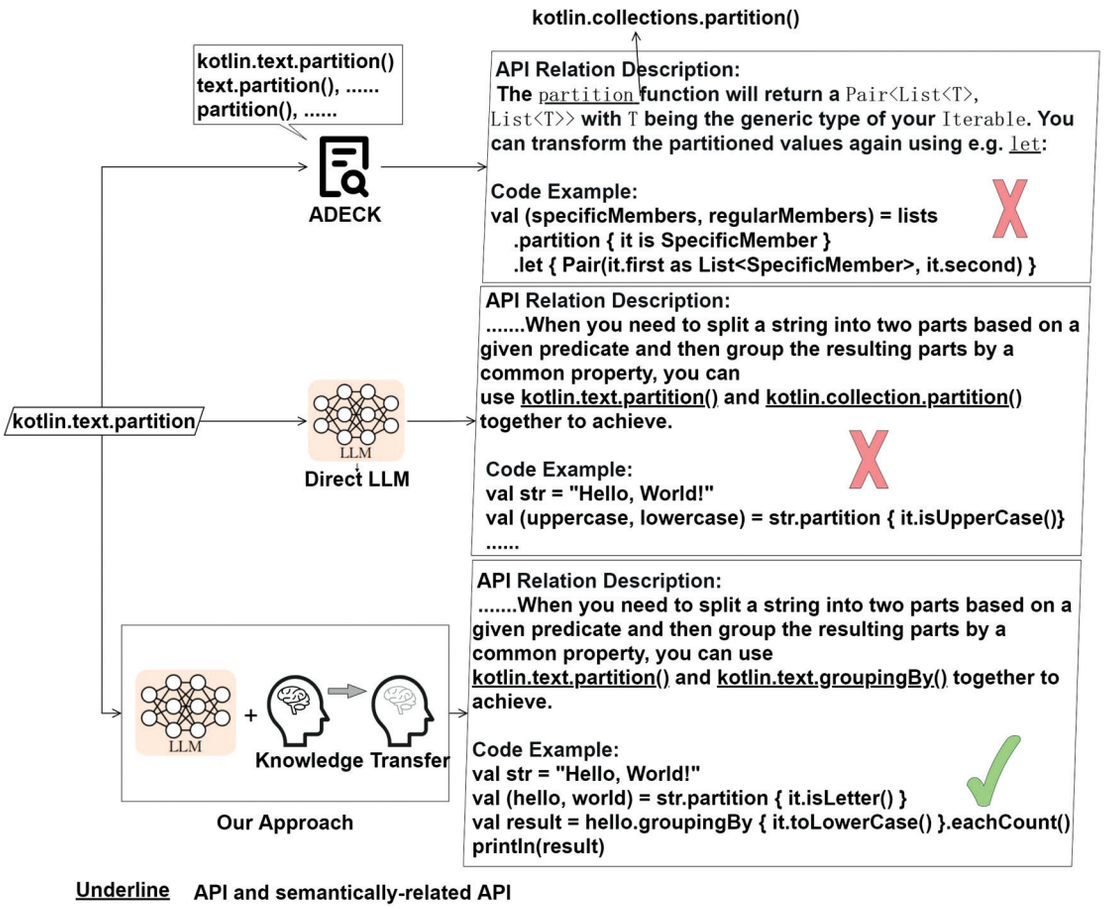

Fig. 11 ADECK vs. direct LLM vs. our approach

through an AI chain. For example, our method generates an on-demand API tutorial with an LLM for *kotlin.text.partition*(), including a code example and a API relation description that reflects the function-collaboration between *kotlin.text.partition*() and *kotlin.text.groupingBy*().

In summary, the experimental results demonstrate the effectiveness of our method in generating API tutorials for languages with limited resources.

3.2 Effectiveness of Our Proposed Strategies

To validate the effectiveness of our proposed strategies including knowledge transfer and hierarchical transfer-behind, we conducted experiments to assess their impact and performance in generating API tutorials. We devise two class variant methods (Non-knowledge transfer method, direct transfer-based method), as described in Section 2.3. Using Dataset-2 as our dataset, we invite two master's students who have participated in the result annotation of Section 3.1 to annotate the results for each variant method.

The results shown in Table 2 demonstrate a significant improvement in success rate when employing the knowledge transfer method (83.75%) compared with the method without knowledge transfer (52.5%). This confirms that knowledge transfer effectively bridges the gap between LLMs and resource-scarce programming languages, enhancing API tutorial generation. Moreover, after employing the hierarchical mapping strategy (up to 83.75%), the success rate has significantly increased compared with the direct mapping strategy (56.25%). Direct mapping often results in multiple APIs. For example, when *java.util.stream.Stream.map*() is mapped to Kotlin, the output is *kotlin.collections.map*(), *kotlin.sequences.map*(), or *kotlin.text.map*(). Instead, the pro-

Table 2 Success rate of different strategies in our approach

%

Method		Success rate
Non-KT		52.50
KT	Direct Transfer	56.25
	Hierarchical-front Transfer	63.75
	Hierarchical-behind Transfer (ours)	83.75
	Hierarchical-both Transfer	71.25

posed hierarchical transfer strategy narrows down the mapping scope, enabling more accurate mapping between resource-rich and resource-scarce ones. Evaluators demonstrated high agreement with kappa[16] values between 88.37% and 100%, indicating strong and consistent consensus.

3.3 Impact of Different Large Language Models on API Tutorial Generation

To explore the differences in the application of various large language models (LLMs) for API tutorial generation, we designed an ablation experiment comparing the number and success rate of tutorials generated by ChatGPT-based method, GPT3.5-based method and GPT4-based method. Using Dataset-2, we enlisted three master's students—who had previously contributed to annotating the results for Section 3.1—to annotate the outcomes for each model. Table 3 compares the number of API documents generated and success rates across different models. The results reveal that the ChatGPT-based method generates 323 tutorials with a success rate of 64.6%. In comparison, the GPT-3.5-based method produces 378 tutorials, achieving a higher success rate of 75.6%. The GPT-4-based method leads the performance, generating 415 tutorials with the highest success rate of 83.0%.

Table 3 Success rate of different models in our approach

Method	Number of supplemented API documents	Success rate/%
ChatGPT-based	323	64.6
GPT3.5-based (ours)	378	75.6
GPT4-based	415	83.0

4 Conclusion

This paper proposes a knowledge transfer method to generate on-demand API tutorials for resource-scarce languages, addressing the scarcity of API documentation in resource-scarce languages. Experiments with 817 Kotlin APIs demonstrate that our approach outperforms the state-of-the-art ADECK method and the gDoc method, significantly improving success rates.

While our work demonstrates strong effectiveness, it is subject to limitations related to suboptimal prompt design and the inherent constraints of the underlying base model. In the future, for suboptimal prompt design, we will conduct dedicated experiments to explore the patterns and principles of effective prompt design for API-related tasks, aiming to optimize the prompt templates for each AI unit in the proposed AI chain. For base model, we plan to adopt advanced large language models with enhanced reasoning capabilities to further optimize the performance of our method.

Data for this study can be found here (https://anonymous. 4open. science/r/On-Demand-Non-Human-Reliant-API-Tutorial-Generation-by-LLM-based-Across-Language-Knowledge-Transfer-96C7/README.md).

References

[1] Huang Q, Yuan Z Q, Xing Z C, *et al*. 1 1>2: Programming know-what and know-how knowledge fusion, semantic enrichment and coherent application[J]. *IEEE Transactions on Services Computing*, 2023, **16**(3): 1540-1554.

[2] Wasserman A I. Software engineering issues for mobile application development[C]//*Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research.* New York: ACM, 2010: 397-400.

[3] Patel P, Cassou D. Enabling high-level application development for the Internet of Things[J]. *Journal of Systems and Software*, 2015, **103**: 62-84.

[4] Baghdadi R, Merouani M, Leghettas M H*et al*. A deep learning based cost model for automatic code optimization[C]//*Proceedings of Machine Learning and Systems.* New York: ACM, 2021, **3**: 181-193.

[5] Ahn B H, Pilligundla P, Yazdanbakhsh A, *et al*. Chameleon: Adaptive code optimization for expedited deep neural network compilation[C]//*International Conference on Learning Representations*. Addis Ababa: OpenReview.net, 2020: 1-17.

[6] Robillard M P, Marcus A, Treude C, *et al*. On-demand developer documentation[C]//2017 *IEEE International Conference on Software Maintenance and Evolution* (*ICSME*). New York: IEEE, 2017: 479-483.

[7] Zhang J X, Jiang H, Ren Z L, *et al*. Enriching API documentation with code samples and usage scenarios from crowd knowledge[J]. *IEEE Transactions on Software Engineering*, 2021, **47**(6): 1299-1314.

[8] Wang S J, Tian Y Q, He D C. gDoc: Automatic generation of structured API documentation[C]//*Proceedings of the ACM Web Conference* 2023. New York: ACM, 2023: 53-56.

[9] Wang Y, Wang W S, Joty S, *et al*. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code under-

standing and generation [C]//*Proceedings of the* 2021 *Conference on Empirical Methods in Natural Language Processing.* Punta Cana: ACL, 2021: 397-400.

[10] Brown T, Mann B, Ryder N, *et al*. Language models are few-shot learners[J]. *Advances in Neural Information Processing Systems*, 2020, **33**: 1877-1901.

[11] OpenAI, Achiam J, Adler S, *et al*. GPT-4 technical report [EB/OL]. [2023-03-15]. https://arxiv.org/abs/2303.08774.

[12] Yang Y R, Zhu Y P, Chen S S, *et al*. API comparison knowledge extraction *via* prompt-tuned language model[J]. *Journal of Computer Languages*, 2023, **75**: 101200.

[13] Bi Z, Chen J, Jiang Y N, *et al*. Codekgc: Code language model for generative knowledge graph construction[J]. *ACM Transactions on Asian and Low-Resource Language Information Processing*, 2024, **23**(3): 2375-4699.

[14] Huang Q, Sun Y B, Xing Z C, *et al*. API entity and relation joint extraction from text *via* dynamic prompt-tuned language model[J]. *ACM Transactions on Software Engineering and Methodology*, 2024, **33**(1): 1-25.

[15] Landis J R, Koch G G. An application of hierarchical kappa-type statistics in the assessment of majority agreement among multiple observers[J]. *Biometrics*, 1977, **33**(2): 363-374.

[16] Nori H, King N, McKinney S M, *et al*. Capabilities of GPT-4 on medical challenge problemse[EB/OL]. [2023-04-12]. https://arxiv.org/abs/2303.13375.

基于大模型驱动的跨语言知识迁移的按需自动化API指南生成

刘智萍

赣东学院 信息工程学院，江西 南昌 344000

摘要：现有的API指南通常仅涵盖单个API的功能描述，缺乏多API间的关系说明及对应的代码示例。基于检索的方法和基于生成的方法可以生成包含API关系说明和代码示例的文档，但受限于API资源的丰富度，在处理Kotlin等资源稀缺型语言时难以发挥效用。为此，本文提出一种面向资源稀缺型语言的按需API指南生成方法：依托AI链架构，将Java等资源富集型语言中的API知识迁移至Kotlin语言。基于500个Kotlin API对该方法进行了实验，结果表明，相较于当前主流的检索式方法ADECK与生成式方法gDoc，本方法生成的API指南数量是ADECK的37倍，gDoc的1.6倍；与未采用知识迁移策略的方案相比，本方法的实施成功率提升了31.25个百分点。上述实验结果证实，借助大语言模型实现跨语言API知识构建具备可行性与应用潜力。

关键词：按需生成；API指南；API关系；大语言模型

□