APIMig: A Project-Level Cross-Multi-Version API Migration Framework Based on Evolution Knowledge Graph

Li Kuang[1] , **Qi Xie**[1] , **HaiYang Yang**[1] , **Yang Yang**[1] , **Xiang Wei**[1] , **HaoYue Kang**[1] and **YingJie Xia**[2,3*]
[1]School of Computer Science and Engineering, Central South University
[2]Micro-Electronics Research Institute, Hangzhou Dianzi University
[3]College of Computer Science and Technology, Zhejiang University
{kuangli, xieqi_, yanghy19, yang978, 8205220302, haoyuekang}@csu.edu.cn, xiayingjie@zju.edu.cn

Abstract

API migration is essential for software maintenance due to the rapid evolution of third-party libraries where API elements may change continuously through updates. There are two main challenges for API migration at the project level, especially across multiple versions: 1) lack of specific library evolution knowledge across multi-version; 2) difficulty in identifying the chain of changes at the project level. This paper proposes a project-level cross-multi-version API migration framework APIMig. We first construct an API evolution knowledge graph (KG) to capture changes between adjacent library versions and then derive coherent cross-version API evolution knowledge by KG reasoning. Second, we design a chain exploration algorithm to track the chain of changes and aggregate the affected code segments. Finally, a large language model is employed in completing API migration by providing the API evolution knowledge and the chain of changes. We construct an evolution KG for the Lucene library from version 4.0.0 to 10.1.0 and evaluate our approach through project migration pairs that depend on different major versions. Our framework shows improvements over the baseline in migrating projects across 7 major versions, achieving average increases of 16.52% in CodeBLEU scores and 28.49% in VCEU scores in GPT-4o.

1 Introduction

In modern software development, third-party libraries are essential for developers, facilitating specific functionalities and enhancing development efficiency for scalable applications [Mahmud *et al.*, 2021]. As technology progresses, third-party libraries continually evolve. Consequently, developers must upgrade these libraries to access new features, fix bugs, and address security vulnerabilities, ensuring project security and compatibility [Kula *et al.*, 2015].

However, library updates typically proceed independently of their callers, and significant changes in new library versions often introduce backward-incompatible API modifications, posing challenges to the quality assurance of upper-level software [Yan *et al.*, 2024]. To prevent compatibility issues arising from library updates, API migration has become a critical task for software quality assurance.

Currently, most migration processes are performed manually by developers, who are required to consult library documentation, version release notes, and open-source community resources to understand new version features, assess their impacts, and estimate the migration effort [Kula *et al.*, 2018]. Developers then manually identify the affected API usage within the project and perform appropriate code replacement to finalize the migration.

To automate the API migration at project-level, especially across multi-version, there are two main challenges:

Challenge 1: lack of specific library evolution knowledge across multi-version. Library maintainers typically document API differences between consecutive versions in change logs or release notes. In practical scenarios, library migrations usually involve skipping intermediate versions. However, the API change information is fragmented since only the changes between adjacent versions are provided. There is a lack of explicit and coherent API evolution knowledge across multiple versions.

Challenge 2: difficulty in identifying the chain of changes at project-level. Due to interdependencies among code elements, the code changes would propagate through the project in a chain, making it challenging to promptly identify and adjust indirectly affected code segments. Consequently, achieving automated code migration at the project level becomes highly challenging.

Traditional approaches [Fazzini *et al.*, 2019; Xu *et al.*, 2019; Nielsen *et al.*, 2021; Lamothe *et al.*, 2022; Wang and Yu, 2022] to API migration involve extracting migration instances from the current and target versions of open-source projects, applying them through pattern matching to find suitable instances. The rapid evolution of large language models (LLMs) has led to research that leverages their strong understanding and generation capabilities for code migration tasks. Existing studies [Almeida *et al.*, 2024; Wu *et al.*, 2024] are typically empirical-oriented, focusing on validating the feasibility of generated migration code by LLMs. Empirical studies show LLMs struggle to meet the

*Corresponding author

challenges, as current models are hard to adapt to changing libraries promptly and to understand library dependencies.

To address the challenges mentioned above, we propose a project-level cross-multi-version API Migration framework (APIMig). To address the specific library evolution knowledge across multi-version (Challenge 1), we first construct an API evolution knowledge graph to capture the dynamic API changes as the version evolves and then derive the coherent cross-version API evolution knowledge by KG reasoning. To address identifying the chain of changes at project-level (Challenge 2), we design a chain exploration algorithm based on the project dependency graph, allowing us to identify the comprehensive code segments impacted by changes throughout the project. Finally, we construct prompts for the API migration requirement, guiding the LLMs to complete the API migration task by providing API evolution knowledge and the chain of changes.

The contributions of this paper are as follows:

1. **Cross-version API evolution knowledge:** We propose a dynamic API Evolution Knowledge Graph (APIEvoKG) and derive coherent multi-version API evolution knowledge by KG reasoning.
2. **Project-level chain of changes:** We propose a Chain Exploration Algorithm(CEA) based on project dependency graph to track the chain of changes and identify all code segments to be updated within the project.
3. **Dataset and effectiveness:** We construct a KG for the Lucene library, and the migration project pairs datasets from different major versions. The experiments demonstrate the effectiveness of our approach in project-level cross-multi-version API migration task.

2 Related Work

2.1 Instance-Based API Migration Method

The instance-based approach to project migration is predominant in existing research, focusing on deriving editing scripts from migration instances and applying them to new contexts. [Nguyen *et al.*, 2010] used a code analysis tool to analyze modifications in API declarations and extracted API usage snippets from clients that successfully migrated. They used a pattern miner to infer adaptation patterns, aiding library version migration automatically. [Xu *et al.*, 2019] proposed Meditor, a template-based migration method in two stages. They sourced projects using the target library from GitHub, analyzed commit histories to extract API migration edits, and generated migrated programs by matching these edits to source code templates. [Lamothe *et al.*, 2022] proposed A3, a migration method based on the principles of mining, applying, and testing migration patterns. A3 extracted migration patterns from various sources using API name matching and data flow graph. Then, these patterns were applied to the target code to facilitate migration, and test suites were used to confirm the effectiveness of migration.

Approaches based on project-level migration instances identify potential changes by mining API migration patterns, which means that their applicability and flexibility are limited [Bai *et al.*, 2024]. Migration relies on programs that have successfully migrated to new library versions, which requires a large number of API usage examples. In practice, finding migration instances for each API change is often infeasible, particularly for these changes in new library versions. Moreover, the differences between versions make it difficult to establish associations between changes.

2.2 LLM-Based API Migration Method

LLMs have been little used in API migrations. [Almeida *et al.*, 2024] introduced a library migration framework, evaluating different prompt types for LLM-based API migration. [Bairi *et al.*, 2024] employed prompt engineering to integrate the migration task instructions with API modifications and contextual information. Meanwhile, [Zan *et al.*, 2022] and [Zhou *et al.*, 2023b] observed that LLMs frequently encounter difficulties in generalizing beyond their training data distribution. They suggested employing Retrieval-Augmented Generation (RAG) to access more efficient APIs, thereby enhancing code generation capabilities. However, [Wu *et al.*, 2024] highlighted the challenges in using RAG for version-controlled code generation, as ambiguous version-related queries can complicate the retrieval process [Wang *et al.*, 2020]. Specifically, similar embeddings of version strings, such as "V2.1.3" and "1.3.2V", can hinder the accuracy of retrieval systems to distinguish version-specific features and functionalities [Zhou *et al.*, 2023a].

In response to the practical needs of cross-version API migration, we propose the APIMig framework. We design a scalable and dynamic API evolution knowledge graph to effectively infer the coherent knowledge of multi-version API evolution. By analyzing the affected elements in the knowledge graph and employing the chain exploration algorithm, we can accurately identify the affected code that needs to be modified, thereby improving the accuracy of migration results generated by LLMs.

3 Approach

Problem Definition. The project-level API migration task across multi-version is defined as follows: Consider a third-party dependent library L, which has n versions $V_1, V_2, V_3, \ldots, V_n$. Given a project P to be migrated, which depends on the version V_i of L, the target is to migrate the code in P to version V_j (only considering upward migration, $1 \leq i < j \leq n$).

Framework Overview. Figure 1 shows the framework of APIMig. APIMig focuses on facilitating project-level cross-multi-version API migration through the use of API evolution knowledge. The framework consists of three key components: *1) API Evolution Knowledge Reasoning.* We built the APIEvoKG to capture the changes that occur between adjacent library versions and derive the coherent cross-version API evolution knowledge through KG reasoning. *2) Chain of Changes Exploration.* We design a chain exploration algorithm to track the code segments affected by API changes through the project dependency graph. *3) LLM-based API Migration.* We employ LLM to facilitate the completion of

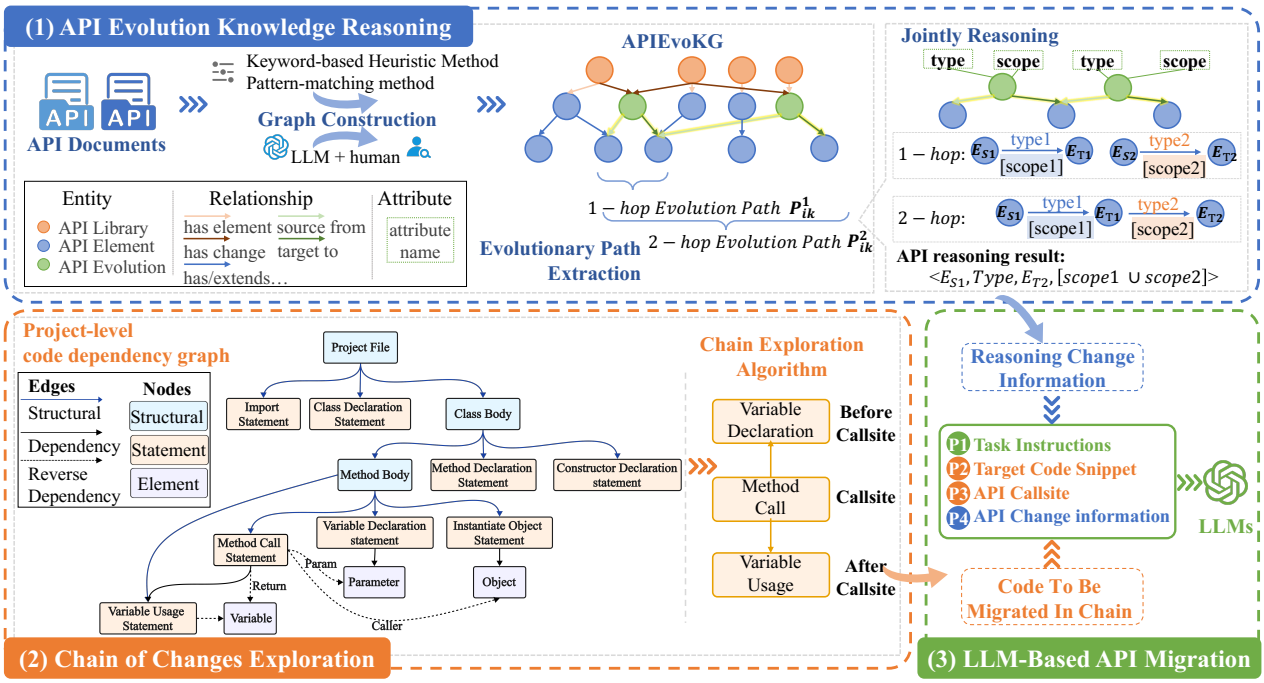

Figure 1: Overview of our framework APIMig

API migration using the API evolution knowledge and the identified chain of changes.

3.1 API Evolution Knowledge Reasoning

Construction of the APIEvoKG. The APIEvoKG schema was specifically designed to capture key elements of the API documentation and track evolution within those APIs.

The entities include three types: API Library (E_L), API Element (E_e), and API Evolution (E_{Evo}). The primary entity-relationship-entity triples are as follows: (E_L, has element, E_e), (E_L, has evolution, E_{Evo}), (E_{Evo}, source from, E_e), and (E_{Evo}, target to, E_e). The "has..." relationship indicates a containment relationship between entities, "source from" indicates the E_e to be evolved, and "target to" indicates the E_e after the evolution. The E_{Evo} entity has two key attributes: "type", which includes add, remove, and update to represent the type of API Evolution, and "scope", which denotes the evolution target (e.g., Method, Parameter).

Based on the aforementioned schema, we develop different extraction methods for types of knowledge from API documents, release notes, and change logs. We employ a keyword-based heuristic approach to extract entities (E_e) such as class and method, as well as the dependency relationship. We use a pattern-matching method to extract fine-grained E_e such as parameters. Furthermore, we use prompts to guide LLMs through a few-shot learning to extract (E_{Evo}) and the relationship between E_{Evo} and E_e. Furthermore, we conduct human reviews to ensure the quality of the extraction process.

API Evolution Knowledge Reasoning. In APIEvoKG, E_{Evo} captures API evolution information between adjacent versions. During cross-version evolution, API elements often experience multiple, successive changes. Inspired by [Zhao *et al.*, 2024], we extract evolutionary paths and infer cross-version API evolution knowledge through reasoning rules.

1) Evolutionary Path Extraction. Define E_{LV_i} as an API evolution entity in version V_i of Library L, E_S as the source API element entity, E_T as the target API element entity, and "type" and "scope" represent attributes of the evolution, respectively. A set of 1-hop evolutionary paths denoted as P^1 can be extracted by the following path (1) and will be appended to the API Evolutionary Path List (EPList$_i^j$).

$$P^1 = \langle \text{“E}_S, \text{Type}, \text{E}_T\text{”}, [\text{Scope}] \rangle \tag{1}$$

To capture the API evolution knowledge of L from version V_i to V_j, while traversing the knowledge graph, if the API element entity in the 1-hop path of the subsequent version is included in the path in the EPList$_i^j$, we extract a 2-hop evolutionary path P^2. By repeating these operations, we continue until all change entities have been traversed, and we obtain the complete set of multi-hop evolutionary path P^n (2) for evolved APIs across multi-version of library L.

$$P^n = \langle \text{“E}_{S1}, \text{Type}_1, \text{E}_{T1}, \text{E}_{S2}, \text{Type}_2, \text{E}_{T2}, \ldots\text{”}, [\text{Scope1}, \text{Scope2}, \ldots] \rangle \tag{2}$$

2) Jointly Reasoning. We conduct joint reasoning for evolutionary paths over 2-hop, which includes reasoning about change operations and their scopes.

Using 2-hop reasoning as an example, in a 2-hop context, two changes can influence different scopes of the same element, such as modifying method parameters and altering its return value type. Therefore, we define a union operation to represent the impact of these scopes: if Scope1 = Scope2, we select Scope1; if Scope1 differs from Scope2, we take their union, indicating that changes have occurred in both scopes.

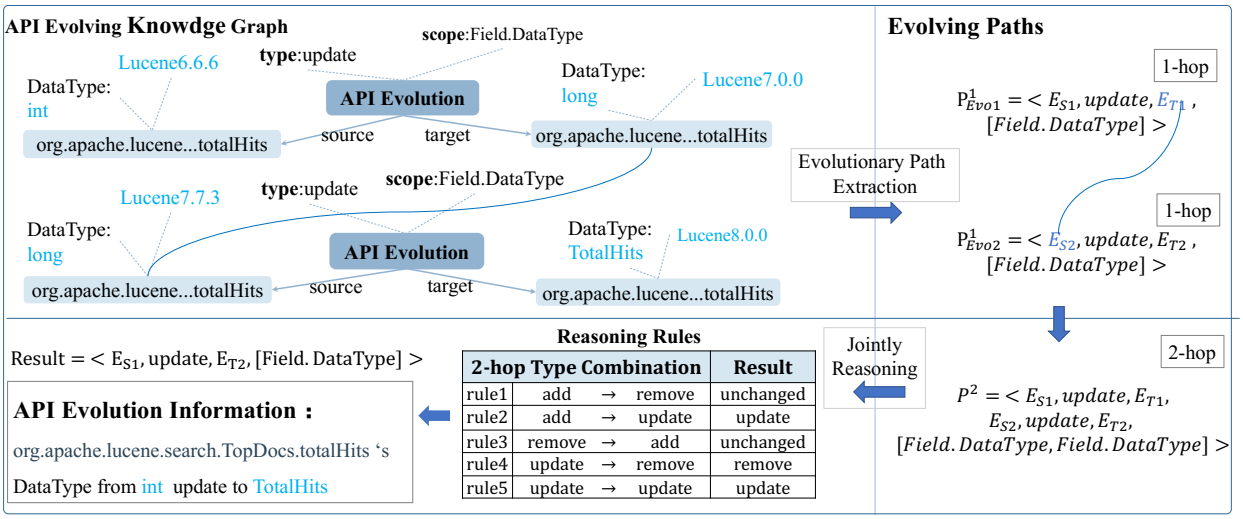

Figure 2: Reasoning Rules and Jointly Reasoning Example

Additionally, we identify three atomic evolution types: *add*, *remove*, and *update*. Different atomic operations within the same scope may have varying impacts. We present the evolution reasoning rules in Figure 2, and provide an Example. In this example, we obtain evolution knowledge for the Lucene Library from version 6.0.0 to 8.0.0 in APIEvoKG, including API Evolution Entities and their related API Element Entities. For E_{Evo1} and E_{Evo2}, we can respectively extract 1-hop evolution paths according path1. Since E_{S2} is included in P^1_{Evo}, we can extract a 2-hop evolutionary path P^2. The two evolution entities share the same scope, and based on rule5 of Reasoning Rules, we can derive the inference result and corresponding description. In this way, we derive the cross-version API change information for evolutionary paths in EPList_i^j and get the valid reasoning results.

3.2 Chain of Changes Exploration

Project-Level Code Dependency Graph. We designed a heterogeneous dependency graph to represent the structure and dependencies of a project.

This graph consists of three types of nodes: 1) *Structural* nodes, such as the class body, which organize the code structure; 2) *Statement* nodes, including class declarations, method declarations, and others, which detail specific code statements; and 3) *Element* nodes, such as variables and parameters, which describe the components of the statements.

The graph includes three types of edges: 1) *Structural* edges (str), connecting *Structural* nodes to *Statement* nodes or between *Statement* nodes, thereby highlighting hierarchical relationships and the execution order between statements; for example, a method body contains multiple statements; 2) *Dependency* edges (dep), representing a forward dependency relationship based on the execution order between nodes; and 3) *Reverse Dependency* edges (redep), indicating a backward dependency between the current *Statement* node and the preceding *Element* node, such as the relationship between the current method call statement and its parameters.

Callsite Analysis. API migration affects a project beyond immediate call sites due to interdependencies between code elements and the impact can significantly vary based on the scope of changes. For instance, modifications to API method parameters and return types can influence different parts of the project. To illustrate this, we define the Callsite Analysis rules in Figure 3 that outlines the potential impact range of various code elements under different change scopes. In this context, we define "Callsite" as the statement where an API is invoked, "Before Callsite" as the preparatory statements preceding the API call (with redep), and "After Callsite" as the statements that utilize the return of the API call (with dep).

Chain Exploration Algorithm. We developed the Chain Exploration Algorithm (CEA) to track the chain of changes and identify all code segments that require updating throughout the entire project, as demonstrated in Algorithm 1.

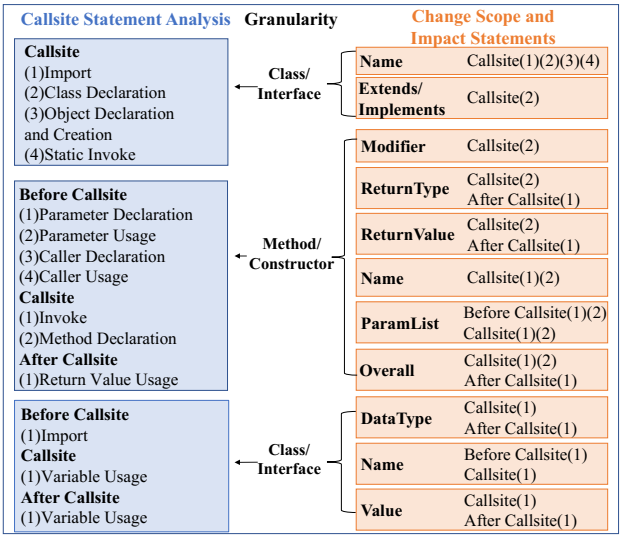

Figure 3: Callsite Analysis

Algorithm 1 Chain Exploration Algorithm

Input: Dependency Relation Graph $G = (N, E)$, Callsite list $N^* = \{n_0^*, n_1^*, \ldots\}$, $N = \{n_i \in G\}$
$ind(n_i)$ return the in-degree of n_i
$getImpactNodes(gran, scope, n_i)$ return Impact Statements Nodes
Output: Callsite Block List $\mathcal{CBL} : \{B_0, B_1, \ldots\}$

while $|N^*| > 0$ **do**
  $n_i \leftarrow argmin_{n_i \in N} ind(n_i)$
  if $n_i \in N^*$ **then**
    $N_{ipt} \leftarrow getImpactNodes(gran, scope, n_i)$
    $\mathcal{B_{CS}}, \mathcal{A_{CS}} \leftarrow [], []$
    for each $n_j \in N_{ipt}$ **do**
      if $e(n_i, n_j).type$ == dep **then**
        $\mathcal{A_{CS}}.append(n_j)$
      end if
      if $e(n_i, n_j).type$ == redep **then**
        $\mathcal{B_{CS}}.append(n_j)$
      end if
      $ind[n_j] \leftarrow ind[n_j] - 1$
    end for
    $\mathcal{CBL}.append(\{\mathcal{B_{CS}}, n_i, \mathcal{A_{CS}}\})$
    $N^*.remove(n_i)$
  end if
  $N.remove(n_i)$
end while
return $\mathcal{CBL}$

We define a **C**all**S**ite **L**ist ($\mathcal{CSL}$) to track change propagation in the dependency graph. We begin by identifying the node with the minimum in-degree. If this node is a Callsite node, we determine the impacted statement nodes from Callsite Analysis based on the API granularity and change scope. Nodes connected by dependency edges are recorded as After Callsite ($\mathcal{A_{CS}}$), while those connected by reverse dependency edges are recorded as Before Callsite ($\mathcal{B_{CS}}$). Consequently, all code segments affected by this Callsite node are aggregated into $\{\mathcal{B_{CS}}, Callsite\ node, \mathcal{A_{CS}}\}$. We then decrement the in-degree of each of the impacted statement nodes by one. Repeat the above steps until all Callsite nodes are processed. This algorithm maintains the order of change propagation and aggregates affected code segments, enabling the LLM to apply changes sequentially in later prompts.

3.3 LLM-Based API Migration

To complete the migration for the project, according to the code segments in the CBL, we construct a prompt for each callsite block of API changes, utilizing API evolution knowledge from the APIEvoKG to guide the LLM in generating the migrated code. The structure of our prompt template is illustrated in Figure 4. We begin with the task definition for project-level cross-version library migration p1; provide the code to be migrated p2. Next, we list the statements that contain calls to changed API elements p3. Finally, we present the information from the APIEvoKG, including source entity, change type, target entity, and change scope p4.

P1 **Task Instructions:** I update my Java project's dependency Library **[Library Name]** from version **[source_version]** to **[target_version]**. I will provide you with the Code Snippet that has API call problems due to version change. Your task is to modify the Code Snippet based on the information given.
P2 **Target Code Snippet:** This is the code that need to migrate, containing the Changed API and Dependent code statements...
P3 **API Callsite:** The statement containing the changed API Callsite...
P4 **API Change information description:** <change source element, type, change target element> [scope],[Description]
Migrate the **"Target Code Snippet"** to target version according the given API Change information. Only return the Migrated Code.

Figure 4: Prompt Template

4 Experiments

4.1 Datasets

Dependency Library. The cross-version API migration task aims to modify the project in response to update in dependency library. We select library for migration based on their current popularity in Maven, the frequency of version changes, and previous research on libraries migration. For our study, we choose Lucene, an open-source information retrieval library. Following the principles of semantic versioning principles [Preston-Werner, 2013; Lam *et al.*, 2020] where major versions indicate breaking API changes while minor/patch versions are used to maintain backward compatibility or bug fixes, we analyzed Lucene's evolution from version 4.0.0 to version 10.1.0 (Oct 11, 2012 - Dec 20, 2024) - encompassing 7 major versions and 118 total subversions - to extract API and change information and construct a comprehensive version evolution knowledge graph.

Dataset for Migration Projects. Previous API migration tasks primarily concentrated on code block-level migrations, lacking comprehensive data on cross-version project migrations. Utilizing the dataset construction methodology detailed in [Zhong and Meng, 2024], we expanded and developed a dataset of API usage example projects in 7 major versions of Lucene, all released by Apache, ranging from version 4.0 to 10.0. Based on these projects, we built 28 migration pairs, involving different version spans, from a span of 0 version to that of 6 versions.

4.2 Metrics

We employ two primary evaluation metrics, CodeBLEU (CB) and Valid Code Element Usage Rate (VCEU), to evaluate our method for handling changes across multi-version libraries in projects.

- CodeBLEU [Ren *et al.*, 2020] evaluates the quality and accuracy of the generated code by comparing its structure, syntax, and vocabulary with the ground truth in the target version. $\text{CodeBLEU} = 0.25 \cdot \text{Match}_{\text{N-gram}} + 0.25 \cdot \text{Match}_{\text{Weighted N-gram}} + 0.25 \cdot \text{Match}_{\text{Syntactic AST}} + 0.25 \cdot \text{Match}_{\text{Semantic data-flow}}$.
- VCEU evaluates the validity of API changes by matching API-related keywords against the ground truth with regular expressions. Score $= \frac{n}{k}$, where n is the number of matched keywords from total keywords k.

Base LLM	Method	span-0		span-1		span-2		span-3		span-4		span-5		span-6	
		CB	VCEU	CB	VCEU	CB	VCEU	CB	VCEU	CB	VCEU	CB	VCEU	CB	VCEU
Claude-3	FR_b	29.06	25.82	27.48	50.24	29.24	44.12	30.14	33.67	30.34	39.42	29.56	31.19	32.66	32.55
	FR_c	26.19	17.50	28.28	52.36	36.23	48.73	38.86	34.01	32.80	35.59	28.29	21.35	22.49	15.75
	VC	32.39	27.10	31.93	52.93	36.25	48.20	39.90	32.66	36.48	41.55	34.53	31.19	40.90	33.27
	CP	36.03	37.87	41.68	**71.47**	40.26	61.11	37.38	47.69	38.53	59.08	35.95	43.27	36.61	47.06
	APIMig	**51.58**	**55.33**	**51.45**	63.72	**62.04**	**76.22**	**66.16**	**84.55**	**62.31**	**88.85**	**58.96**	**80.37**	**51.73**	**67.58**
Deep Seek-V3	FR_b	34.39	37.01	36.18	74.69	37.84	74.57	43.56	75.18	38.85	71.50	36.76	60.43	39.31	65.46
	FR_c	31.61	26.33	30.57	44.15	33.93	37.27	30.58	31.56	29.52	37.76	31.46	33.18	32.95	35.72
	VC	42.37	44.67	42.22	71.60	45.86	77.34	52.25	63.72	46.53	71.09	44.85	54.93	46.65	68.40
	CP	46.64	71.11	49.47	69.84	51.25	85.75	50.83	84.07	48.47	83.01	45.70	71.98	46.34	73.92
	APIMig	**64.13**	**88.84**	**63.99**	**73.14**	**68.19**	**90.09**	**70.01**	**90.89**	**62.99**	**89.76**	**61.02**	**80.90**	**59.39**	**80.98**
GLM-4	FR_b	33.21	20.83	30.78	48.86	35.11	49.72	41.13	49.57	36.87	51.39	34.12	38.33	37.12	41.76
	FR_c	34.40	27.67	33.92	48.74	39.92	38.43	38.93	34.00	36.94	39.91	34.51	34.78	34.65	33.59
	VC	41.75	51.03	41.25	68.59	45.21	76.26	51.57	78.45	46.64	78.29	44.37	67.47	44.59	69.31
	CP	46.09	55.31	50.60	**74.59**	50.55	81.20	50.65	83.52	49.12	81.92	46.03	69.20	44.33	67.84
	APIMig	**59.03**	**69.29**	**57.60**	61.81	**64.87**	**83.80**	**67.49**	**89.10**	**64.32**	**89.22**	**61.40**	**82.35**	**59.98**	**83.59**
GPT-4o	FR_b	34.74	30.53	32.06	50.48	34.84	48.41	40.74	41.47	36.19	49.68	35.17	38.01	36.61	44.77
	FRc	31.69	26.98	27.49	47.40	31.76	32.80	34.10	30.90	31.19	39.23	30.80	31.37	33.37	27.06
	VC	38.42	28.53	34.28	51.69	39.44	47.57	43.46	41.81	41.63	49.50	39.02	37.64	39.79	41.47
	CP	44.70	45.23	45.52	53.37	46.64	51.20	45.05	43.00	44.15	51.71	42.16	42.00	40.18	49.61
	APIMig	**59.33**	**64.77**	**54.04**	**58.82**	**62.49**	**78.02**	**66.63**	**84.69**	**64.46**	**89.76**	**60.14**	**80.46**	**56.96**	**79.02**

Table 1: Results of Base LLMs for Different Methods. "span-x" denotes a migration pair's version span (e.g., 4.x.x-5.x.x is categorized as a span-1). The metric for each span is computed as the average of all corresponding migration pairs within that span category.

4.3 Baselines

We compare our results with baseline methods that utilize LLMs, including Automatic Library Migration (FR), VERSICODE, and CodePlan, using the aforementioned datasets.

- **FR**[Almeida *et al.*, 2024]: This method introduces a set of prompts designed to guide LLMs in generating migration code. We compare two types of prompts: FR_b(Zero-Shot) and FR_c(Chain of Thoughts).
- **VERSICODE(VC)**[Wu *et al.*, 2024]: This approach introduces the version-aware code migration task and provides prompts to validate the model's ability to migrate code between specified versions.
- **CodePlan(CP)**[Bairi *et al.*, 2024]: This model tackles project-level code editing tasks by providing contextual information within a class to aid in generating code.

4.4 Implementation Setup

We employ LLMs to accomplish API migration tasks. For the selection of LLMs, we choose the most popular LLMs: *Claude-3*, *DeepSeek-V3*, *GLM-4*, and *GPT-4o*. Our model parameter settings are configured as follows: the *Temperature* is set to 0.7 to strike a balance between text diversity and coherence during the generation process, while *Max Tokens* is capped at 512 to ensure the readability of the code fragments. Additionally, to enhance the robustness of our experiments, we allow each LLM to generate results independently 5 times, reporting the average in our paper.

4.5 Main Results

Table 1 presents the comparison between APIMig and baseline methods across four LLMs. Our approach achieves superior performance in cross-multi-version API migration tasks.

Taking GPT-4o as an example, APIMig achieves a 16.52% average improvement over the state-of-the-art method CodePlan in the CodeBLEU metric for all spans. Specifically, for small version spans (span $\leq$ 2 major versions), the improvement is 13%, while for large spans (span $\geq$ 3 major versions), it rises to 19.16%, particularly reaching 21.58% for span-3. Compared to FR_b, FR_c, and VC, APIMig shows average improvements of 24.81%, 29.09%, and 21.14% in 6 spans, respectively. Furthermore, in the VCEU metric, APIMig achieves a 28.49% average improvement for all spans, including a remarkable 36.90% improvement for large spans. Against FR_b, FR_c, and VC, it outperforms 33.17%, 42.83%, and 33.9% on average in 6 spans respectively.

The results show the superiority of our approach over traditional prompting and chain-of-thought methods in API migration, showing its ability to generate critical code elements. Additionally, experiments on Claude-3, DeepSeek-V3, and GLM-4 further verify APIMig's robustness, demonstrating its adaptability to diverse LLMs.

4.6 Ablation Studies

As shown in Figure 5, we conducted an ablation study on APIMig to investigate the contributions of its key components: (1) removing API evolution information from the knowledge graph (w/o KG), and (2) eliminating code segments in the change chain (w/o Change Chain Impact).

Our results demonstrate that both components are essential for optimal performance. Without the API Evolution Knowledge Graph, LLMs relies only on the training corpus to generate migration code, causing a 33.08% average VCEU metric drop in different spans. The VUEC metric decline confirms that missing API information significantly limits accurate and relevant migration code generation, highlighting the critical role of APIEvoKG in providing version-specific API knowledge.

Without Chain of Changes Exploration, CodeBLEU and VCEU decrease by 22.98% and 5.35%, respectively, demonstrating that third-party API changes propagate through de-

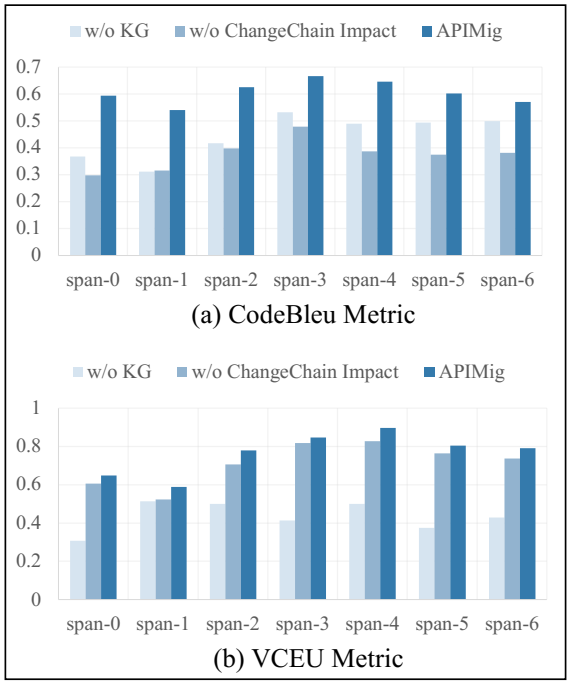

Figure 5: Component Effects on GPT-4o

pendency chains rather than occurring in isolation. This performance drop reveals the model's difficulty in accurately identifying modification locations without understanding contextual code dependencies, highlighting the necessity of dependency information for effective migration.

4.7 Case Study

Figure 6 demonstrates our solution during an upgrade of Lucene in the project from version 4.10.4 to 7.3.1. We utilize our APIEvoKG to generate a cross-version API Change List through reasoning. Then, we build a dependency graph for the project code that requires migration and apply the chain exploration algorithm to pinpoint the affected code that needs updates throughout the project. For instance, the API change on line 128 modifies the invoking method's name and the return value type, which in turn affects line 222, where a function declared on line 115 is invoked. Finally, we use a prompt to guide the LLM to generate the result. Compared to baseline methods, our method correctly utilizes the change information and applies the appropriate contextual content to generate migration code.

5 Threats to Validity

This study proposes a project-level code generation method for migrating APIs across versions using an APIEvoKG. While our method shows significant capabilities, it faces limitations, such as the scarcity of relevant datasets for third-party libraries and open-source projects[Liu *et al.*, 2023]. Our knowledge graph relies on official documentation and change logs, which may have inconsistencies and missing information, affecting change reasoning accuracy. Additionally, automated entity extraction and relationship identification may

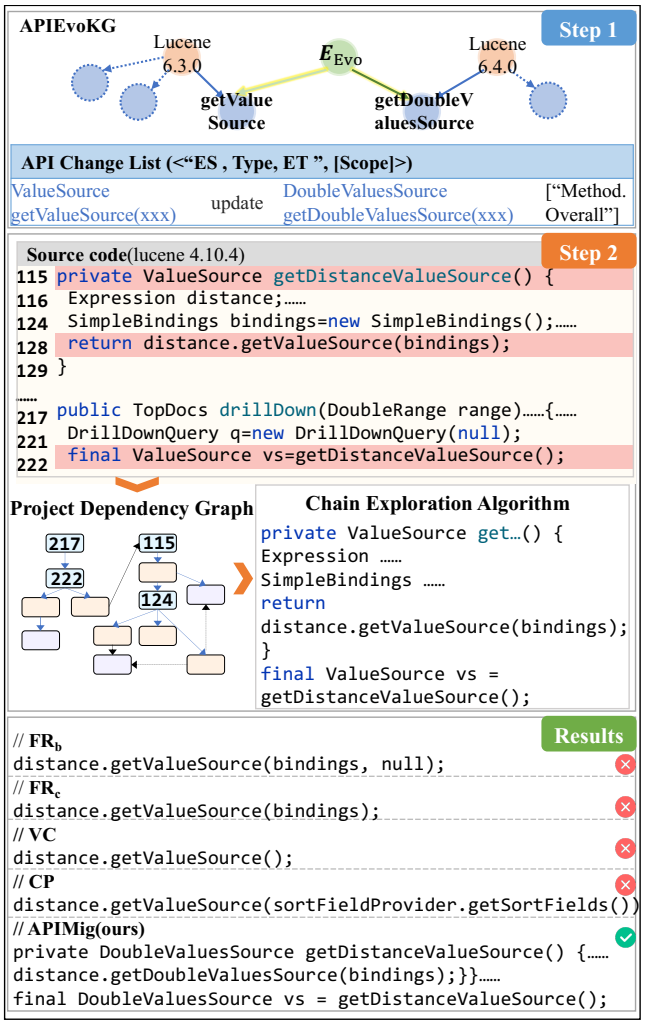

Figure 6: Case Study. Migrate project from lucene 4.10.4 to 7.3.1.

introduce errors, especially with complex changes. To mitigate this, we have manually reviewed the extraction results to minimize errors.

6 Conclusion

In this work, we propose a cross-multi-version API migration framework (APIMig), which provides cross-version evolution knowledge of APIs in libraries through the reasoning of the API Evolution Knowledge Graph and locates the impact scope of API changes in the project code using a chain exploration algorithm. Based on a large language model, it facilitates the migration of code within the project. This approach dynamically expands the cross-version API knowledge of the LLMs while precisely identifying the code that needs to be modified in context, thus enabling API migration. Experiments conducted on cross-version projects demonstrate the effectiveness of our proposed method, which achieves competitive results across various cross-version pairs, proving the method's effectiveness and robustness in cross-version library migration projects.

Acknowledgments

This work has been supported by the National Natural Science Foundation of China under grant No.62472447 and 62472132, Hunan Provincial Natural Science Foundation of China under grant No.2024JK2006, the Science and Technology Innovation Program of Hunan Province under grant No.2023RC1023, Key R&D Program Project of Zhejiang Province under grant No.2025C01063 and 2024C01179, the Fundamental Research Funds for the Central Universities of Central South University. This work was carried out in part using computing resources at the High Performance Computing Center of Central South University.

References

[Almeida *et al.*, 2024] Aylton Almeida, Laerte Xavier, and Marco Túlio Valente. Automatic library migration using large language models: First results. In Xavier Franch, Maya Daneva, Silverio Martínez-Fernández, and Luigi Quaranta, editors, *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2024, Barcelona, Spain, October 24-25, 2024*, pages 427–433. ACM, 2024.

[Bai *et al.*, 2024] Weiheng Bai, Keyang Xuan, Pengxiang Huang, Qiushi Wu, Jianing Wen, Jingjing Wu, and Kangjie Lu. Apilot: Navigating large language models to generate secure code by sidestepping outdated api pitfalls. *arXiv preprint arXiv:2409.16526*, 2024.

[Bairi *et al.*, 2024] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram K. Rajamani, Balasubramanyan Ashok, and Shashank Shet. Codeplan: Repository-level coding using llms and planning. *Proc. ACM Softw. Eng.*, 1(FSE):675–698, 2024.

[Fazzini *et al.*, 2019] Mattia Fazzini, Qi Xin, and Alessandro Orso. Automated api-usage update for android apps. In Dongmei Zhang and Anders Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 204–215. ACM, 2019.

[Kula *et al.*, 2015] Raula Gaikovina Kula, Daniel M. Germán, Takashi Ishio, and Katsuro Inoue. Trusting a library: A study of the latency to adopt the latest maven release. In Yann-Gaël Guéhéneuc, Bram Adams, and Alexander Serebrenik, editors, *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pages 520–524. IEEE Computer Society, 2015.

[Kula *et al.*, 2018] Raula Gaikovina Kula, Daniel M. Germán, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? - an empirical study on the impact of security advisories on library migration. *Empir. Softw. Eng.*, 23(1):384–417, 2018.

[Lam *et al.*, 2020] Patrick Lam, Jens Dietrich, and David J Pearce. Putting the semantics into semantic versioning. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 157–179, 2020.

[Lamothe *et al.*, 2022] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Peter Chen. A3: assisting android API migrations using code examples. *IEEE Trans. Software Eng.*, 48(2):417–431, 2022.

[Liu *et al.*, 2023] Mingwei Liu, Yanjun Yang, Yiling Lou, Xin Peng, Zhong Zhou, Xueying Du, and Tianyong Yang. Recommending analogical apis via knowledge graph embedding. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1496–1508, 2023.

[Mahmud *et al.*, 2021] Tarek Mahmud, Meiru Che, and Guowei Yang. Android compatibility issue detection using api differences. In *2021 IEEE international conference on software analysis, evolution and reengineering (SANER)*, pages 480–490. IEEE, 2021.

[Nguyen *et al.*, 2010] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to API usage adaptation. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 302–321. ACM, 2010.

[Nielsen *et al.*, 2021] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Semantic patches for adaptation of javascript programs to evolving libraries. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 74–85. IEEE, 2021.

[Preston-Werner, 2013] Tom Preston-Werner. Semantic versioning 2.0.0, 2013.

[Ren *et al.*, 2020] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[Wang and Yu, 2022] Kai Wang and Ping Yu. Augraft: Graft new API usage into old code. In *Internetware 2022: 13th Asia-Pacific Symposium on Internetware, Hohhot, China, June 11 - 12, 2022*, pages 55–64. ACM, 2022.

[Wang *et al.*, 2020] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. An empirical study of usages, updates and risks of third-party libraries in java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–45. IEEE, 2020.

[Wu *et al.*, 2024] Tongtong Wu, Weigang Wu, Xingyu Wang, Kang Xu, Suyu Ma, Bo Jiang, Ping Yang, Zhenchang Xing, Yuan-Fang Li, and Gholamreza Haffari. Versicode: Towards version-controllable code generation. *CoRR*, abs/2406.07411, 2024.

[Xu *et al.*, 2019] Shengzhe Xu, Ziqi Dong, and Na Meng. Meditor: inference and application of API migration edits. In Yann-Gaël Guéhéneuc, Foutse Khomh, and Federica Sarro, editors, *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 335–346. IEEE / ACM, 2019.

[Yan *et al.*, 2024] Jiwei Yan, Jinhao Huang, Hengqin Yang, and Jun Yan. Exception-aware lifecycle-model construction for framework apis. *CHINESE JOURNAL OF COMPUTERS*, 47(09):1989–2008, 2024.

[Zan *et al.*, 2022] Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Yongji Wang, and Jian-Guang Lou. When language model meets private library. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 277–288. Association for Computational Linguistics, 2022.

[Zhao *et al.*, 2024] Ruilin Zhao, Feng Zhao, Long Wang, Xianzhi Wang, and Guandong Xu. Kg-cot: Chain-of-thought prompting of large language models over knowledge graphs for knowledge-aware question answering. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence (IJCAI-24)*, pages 6642–6650. International Joint Conferences on Artificial Intelligence, 2024.

[Zhong and Meng, 2024] Hao Zhong and Na Meng. Compiler-directed migrating api callsite of client code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, 2024.

[Zhou *et al.*, 2023a] Bingzhe Zhou, Xinying Wang, Shengbin Xu, Yuan Yao, Minxue Pan, Feng Xu, and Xiaoxing Ma. Hybrid api migration: A marriage of small api mapping models and large language models. In *Proceedings of the 14th Asia-Pacific Symposium on Internetware*, pages 12–21, 2023.

[Zhou *et al.*, 2023b] Shuyan Zhou, Uri Alon, Frank F. Xu, Zhengbao Jiang, and Graham Neubig. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.