Can LLMs Update API Documentation?

*API Documentation Updates By Augmenting LLMs with Code Changes

Seonah Lee
Software Engineering
Gyeongsang National University
Jinju, Republic of Korea
saleese@gnu.ac.kr

Jueun Heo
AI Convergence Engineering
Gyeongsang National University
Jinju, Republic of Korea
juandeun@gnu.ac.kr

Katherine R. Dearstyne
Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, USA
kdearsty@nd.edu

***Abstract*—Human-written API documentation often becomes outdated, requiring developers to update it manually. Researchers have proposed identifying outdated API references in documentation, yet have not addressed updating API documentation. Now, emerging large language models (LLMs) are capable of generating code examples and text descriptions. Then, a key question arises: Can LLMs assist in updating API documentation? In this paper, we propose an approach for leveraging an LLM to update API documentation with code change information. To evaluate this approach, we select five open-source projects that manage documentation revisions on GitHub and analyze the differences in documentation between two releases to derive ground truths. We then assess the accuracy of LLM-generated updates by comparing them to the ground truths. Our results show that LLM-generated updates achieve higher METEOR than outdated API documentation (0.771 vs 0.679). It indicates that the LLM updates are more similar to the human updates than the outdated documentation. Our results also reveal that LLMs update code-related information in API documentation with a maximum F1 score of 0.921.**

***Index Terms*—API documentation, updates, LLMs, code changes, code summarization**

I. Introduction

API documentation serves as the definitive resource that developers rely on to understand, adopt, and effectively use APIs [1], [2]. To support their development, various tools, such as JavaDoc [3] and Doxygen [4], have been created to automatically generate documentation from code comments. Despite this, API documentation is commonly outdated [5], with studies showing that up to 82.3% of projects have experienced outdated documentation at least once during their history [6], [7]. This is a significant challenge in software documentation, with research showing that outdated information accounts for a substantial portion of documentation issues [8]. While this problem is particularly common in human-written API documentation [2], it remains essential, as it offers users a more concise and practical way to understand and use an API compared to automatically generated alternatives.

Given the impact of outdated API documentation, researchers have explored various methods to address this issue [2], [5]–[7], [9]. For instance, Lee et al. proposed automatically detecting outdated API names and suggesting updated names based on revision histories of source code [2]. Similarly, Tan et al. used regular expressions to find API names in documentation and use these matches to identify outdated API references in documentation [6], [7]. While these approaches have improved the detection of outdated API documentation, there has been limited research on methods for updating the documentation itself.

Emerging large language models (LLMs) offer advanced capabilities for generating documentation, including code examples and textual descriptions. Consequently, researchers have explored using LLMs to enhance API documentation [10] or leveraging API documentation to improve LLM performance [11]. Additionally, studies have investigated the use of LLMs for automatically generating code comments and commit messages [12] or for generating documentation from source code [13]. However, none of these studies have specifically addressed updating human-written API documentation.

To address this gap, we propose updating API documentation by augmenting LLMs with code changes. This approach allows developers to rely on accurate documentation throughout the product lifecycle while reducing the time and effort required to keep them up-to-date. The proposed approach consists of three phases. The first phase is to extract API change rules from code revision histories. The second phase is to identify outdated API references with the API change rules. The third phase is to update document snippets with LLM prompts and API change rules.

We evaluated our approach through both quantitative and qualitative analyses. First, we compared manual updates across five projects with those generated by our method. Given a change rule, GPT-4o and Claude accurately updated API references with F1 scores of 0.921 and 0.900, respectively. GPT-4o and Claude also updated document snippets with METEOR values of 0.771 and 0.759, respectively. We verified that the updates made by GPT-4o and Claude are significantly different from outdated ones. Furthermore, we gathered feedback from developers on the usefulness of the updated documentation. Our results indicated that accurate and concrete updates are crucial for effective reporting.

Our contributions are as follows:

- We propose an approach to update API documentation by augmenting LLMs with API change rules.
- We demonstrate experimental results that LLMs can generate API document updates similar to document updates

by humans than outdated API documents.
- We make the benchmark data for outdated API documents and their ground truth updates available for follow-up researchers. [1]

The remaining parts of the paper are organized as follows. Section II discusses the related work. Section III introduces our proposed approach. Section IV explains our experimental setup. Section V reports our experimental results. Section VI discusses the qualitative results of our experiments, and Section VII concludes our paper.

II. Related Work

There are three research groups relevant to our study. The first group identified outdated API references in API documentation. The second group associated LLMs with API documentation. The third group used LLMs for code changes.

A. Studies for outdated API documentation

The first group focused on identifying outdated API references. Zong and Su developed DocRef to detect API documentation errors, including outdated API references. They compared API references with the latest version of API names and found mismatches between them [5]. Dagenais and Robillard developed AdDoc to detect and recommend updates to documentation based on changes in the codebase. They recommended new code elements that should be documented as well as API references to deprecated or deleted code elements [9]. Lee et al. developed FreshDoc to updated human-written API documentation. They identified outdated API names and made suggestions updated API names based on change rules mined in revision histories of code [2]. Tan et al. searched outdated API names in release notes and read me files by using regular expressions for JavaScript, Java, Python and Go [6]. Tan et al. developed a GitHub Actions tool that alerts GitHub developers outdated code element references in read me files and wiki pages whenever a pull request is submitted [7]. However, they have not yet focused on updating API documentation by leveraging an LLM.

B. Studies of LLMs for API documentation

The second group used LLMs for API documentation or used API documentation for LLMs. Yang et al. suggested APIDocBooster by leveraging GPT-4. The features of APIDocBooster are 1) obtaining the inputs from StackOverflow as well as YouTube, and 2) connecting extractive summaries and abstract summaries in pipelines [10]. Khan and Uddin proposed generating code documentation by using OpenAI's Codex. In prompt they put source code and asked to generate the code description as sentences [14]. Su et al. explored the use of LLMs to extract valuable information such as locking rules, exception predicates, and performance-related configurations from API documentation [15]. Lazar et al. proposed SpeCrawler to transfer REST API documentaiotn into an Open API specification that represents data formats for RESTful APIs by using LLMs [16]. Wang et al. proposed two prompt-fix based approaches for deprecated API usage in LLM-based code completion. The first one is to replace deprecated API tokens then regenerate them. The second one is insert additional replacing prompts then regenerate them. [17]. Jain et al. created DAC, a database of a bunch of API documentation to augment LLMs. They discovered that DAC was useful for the APIs with low frequency but not for the APIs with high frequency, and revised DAC to DAC++ that does not retrieve the database with the APIs with high frequency [11]. However, they have not paid attention to updating API documentation with LLMs.

C. Studies of LLMs for code changes

The third group applied LLMs to code changes. Fan et al. conducted an empirical study to explore the capabilities of LLMs on code change related tasks such as generating code reviews, generating commit messages, and updating comments just-in-time [12]. Yu et al. proposed a fine-tuned Large Language Model (LLM) designed to enhance accuracy and comprehensibility in automated code review. For that, they collected GitHub review comments and fine-tuned open-source LLMs (e.g., Llama) with the collected reviews [18]. Imani et al. investigated whether open-source Large Language Models (OLLMs) can replace proprietary models like GPT-4 for commit message generation (CMG) and showed that OLLMs surpasses GPT-4 [19] Zhang et al. proposed generating commit messages with LLMs and RAG technologies. Given a diff in a commit, their approach first retrieve the most similar diff and its commit message and combines the original diff, the retrieved diff and the commit messaged. Last, their approach asks LLMs to generate commit message with the combined inputs [20]. Li et al. also proposed generating commit messages with LLMs and various other information around a commit [21]. Zhang et al. proposed using LLMs to generate parameter constraints from code comments and applying static program analysis to code with the extracted parameter constraints [22]. However, they have not addressed the issues of updating API documentation.

III. Proposed Approach

We propose an approach to update API documentation by utilizing an LLM with API change rules. As shown in Figure 1, the proposed approach consists of three phases. The first phase extracts API change rules from a code revision history. The second phase finds outdated API references in API documents by utilizing the API change rules. The third phase updates API documents with LLMs, around the outdated API references.

A. Extracting API Change Rules

Phase 1 extracts API change rules from code revision histories, similar to Step 1 (Extracting Change Rules) of FreshDoc [2]. The API change rules will be used in identifying outdated API references in documentation in Step 2 and retrieving change information in Step 3.

[1] https://zenodo.org/records/15022935

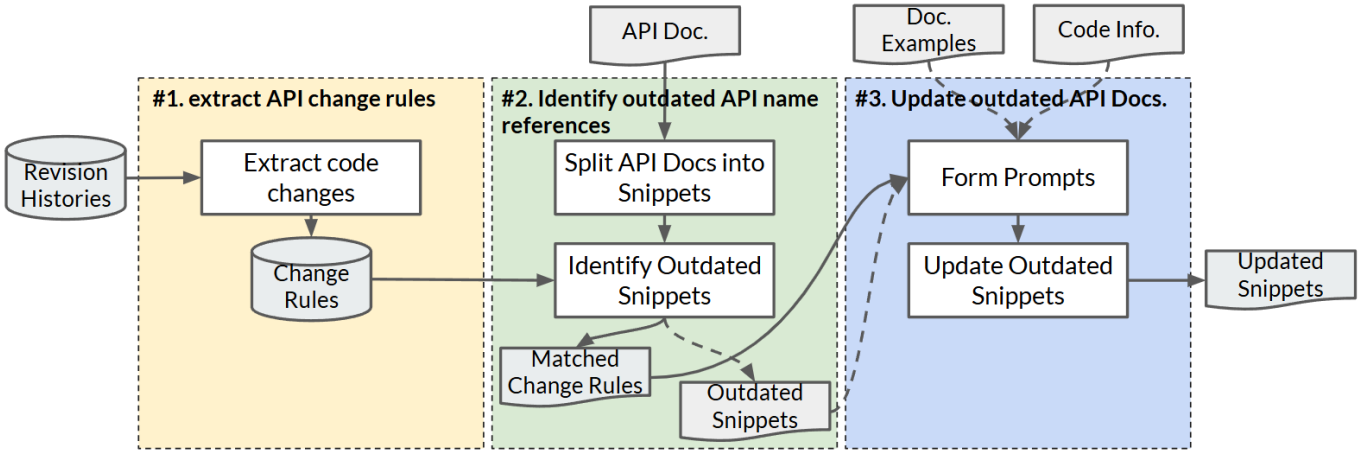

Fig. 1: Overview of the Proposed Approach

The input of the phase is the revision histories of a target project RH, including the histories of any external libraries that the target project uses. The phase extracts each revision rh_n from the software revision history and retrieves each source file f_n within that revision. It then compares the previous and current versions of each source file, f_{n-1} and f_n, respectively. Based on the comparison, the step derives change information about an API name $c(e_{n-1}, e_n)$.

The output of the step is a set of API change rules. A rule can be expressed as a tuple (t, e_{n-1}, e_n), where t represents a commit id, e_{n-1} an outdated API name, e_n a new API name. The change rules are related to modifications, deletions, and additions of API names.

- **Modifications** involve changes to either the simple name or the entire qualified name (i.e., the changes to a package name). A modification can be expressed as (t, e_{n-1}, e_n), where both e_{n-1} and e_n are not null.
- **Deletions** occur when classes, methods, or fields are removed from a file. If the file is deleted, it exists in the previous version but not in the next version. A deletion can be expressed as $(t, e_{n-1}, \varnothing)$, where $\varnothing$ is null.
- **Additions** occur when new classes, methods, or fields are introduced in a file. If a file is added, it does not exist in the previous version but exists in the next version. An additions can be expressed as $(t, \varnothing, e_n)$.

These API change rules govern the renaming, adding, and deleting of API names. For example, an API change rule $(2ecb4e,\ ...MyDataSourcesConfiguration,\ ...MyAdditionalDataSourceConfiguration)$ can represent the renaming of $...MyDataSourcesConfiguration$ to $...MyAdditionalDataSourceConfiguration$.

To implement this phase, we reviewed several tools that analyze the revision histories [2], [23], [24]. Considering the time it takes to extract change rules, we chose a less time-consuming automated method [2].

B. Identifying Outdated API References

Phase 2 leverages the API change rules extracted in Phase 1 to identify outdated API references in the documents. The main inputs to this phase are the API documents and change rules. The output are the outdated sections as well as the change rules matched to those sections. It consists of two steps, as follows.

1) Splitting API documents into document snippets: This step splits API documents into API sections and then further breaks it down into document snippets, if needed. This allows for creating smaller, more manageable units that are easier to update.

The input of the step is a set of API documents D, where each API document d_i is a file written in a markup language[2] First, this step identifies section headings by counting the number of #'s or ='s and then splits the content of a file by the section headings. The output of the step are API sections s_j, where an API section is a file that is split from the content of an API document d_i. The file name of each API section is created by appending the original file name to the word "section" and concatenating the title of each section.

If a section remains quite long, more than 100 lines, simply splitting the document into multiple sections may not be sufficient. In such cases, a section can be further split based on the trace links of the code elements in the API section. For example, the tag $include - code$ ::[3] can be used as a criterion for splitting an API section. If a section s_j includes multiple tags linking code elements to the API documentation, it can be split accordingly. The resulting smaller units, which are extracted from the API document and used in the next step, are referred to as document snippets ds_k.

2) Identifying outdated document snippets: This step identifies API document snippets that contain outdated API references. The inputs are the API snippets and change rules.

[2]Specifically, we assume a markup file handled by the asciidoctor tool; the tool helps formatting human-written API documentation.

[3]The tag is one of the tags used in the Asciidoctor tool.

First, each word in each API snippet ds_k is checked against the simple name of an outdated API name e_{n-1} in the API change rules. For modification and deletion cases, the step verifies if all of the terms in the qualified name of e_{n-1} are present in the API section. If so, it checks if the qualified names are outdated.

If there are several candidate qualified names, the step calculates the the sum of distances between the terms in the qualified name of e_{n-1} and the word that is being examined in the document snippet. The qualified name with the shortest distance is selected. The formula that calculates the distance is described in an earlier paper [2].

Addition cases require a different approach since there is no outdated API reference to update. In this case, the step first locates a a source file that is linked to the API names referred to in the API document d_i. It then identifies the methods and the fields that are not documented unlike other methods or fields in the same source file at the method or field level. Next, the step determines which method or field that should be added, and finally, it identifies where the section should be added according to the order of the methods or fields in the source file f_n.

The final outputs are the outdated sections matched to the associated API change rules.

C. Updating outdated API Documents

Phase 3 leverages an LLM to update the document snippets using a given API change rule. To accomplish this, the phase consists of two steps, as follows.

1) Forming prompts: This step constructs a prompt to update the API document snippets. The inputs include a document snippet containing outdated API references and the matching change rules. Optional inputs may include API document examples and the relevant source code information, as is the case for additions.

> [**API change rule for modification**]
> *In the spring-boot project, the method ...MyDataSourcesConfiguration was modified to ...MyAdditionalDataSourceConfiguration.*
>
> [**Outdated API document snippet**]
> *Here, the outdated document relevant to the modified code exists:*
> *Configure Two DataSources: ...*
> *include-code::MyDataSourcesConfiguration[] ...*
>
> [**Instructions**]
> *Can you update the document in asciidoc format?*
> *- Produce only the updated document.*
> *- Maintain the same format as the given document.*
> *- Please do not put the code itself. Keep the original format.*
> *- Keep the same sentences as much as possible if the outdated document exists.*

Different prompts are constructed for modification and deletion versus addition cases. For modifications and deletions, outdated API document snippet ds_k and the corresponding API change rule (t, e_{n-1}, e_n) are used as input. As shown in the example above, the prompt consists of three parts: an API change rule related to the update, an API document snippet that needs to be updated, and instructions for updating the document snippet.

In the case of additions, since there is no outdated, original API document snippet, an API document example is provided. Code information is also included, because the addition case involves creating a new API document snippet for the added code. This helps the LLM understand the documentation style of a project.

As shown below, the prompt consists of four parts: an API change rule related to the update, code information relevant to the addition, an exampled API document snippet, and instructions for generating the document snippet. The prompt includes the code information corresponding to the API change rule at the method or field level. It also includes the examples of API document snippets, typically the first document snippet ds_k that belongs to the same document d_i.

> [**API change rule for addition**]
> *In the elasticsearch-java project, the method co.elastic.clients.documentation.usage.IndexingBulkTest.useBulkIndexer() was added.*
>
> [**Code information**]
> *The following is the test code:*
>
> ```
> @Test
> public void useBulkIndexer() throws Exception
> { ...
>     //tag::bulk-ingester-setup
>     BulkIngester<Void> ingester = BulkIngester.of(b
>          -> b
>         .client\(esClient\)    ...
>         .maxOperations(100)  ...
>         .flushInterval(1, TimeUnit.SECONDS) ...
>     ); ...
>     ingester.close(); ...
>     //end::bulk-ingester-setup
> }
> ```
>
> [**Exampled document snippet**]
> *Here, the exampled document exists: ...*
> *==== Indexing application objects ...*
> *[“source”,“java”] ...*
> *include-tagged::{doc-tests-src}/usage/IndexingBulkTest.java [bulk-objects] ...*
> *<1> Adds an operation ...*
>
> [**Instructions**]
> *Can you generate the document relevant to the added code in asciidoc format?*
> *- Produce only the updated document.*
> *- Maintain the same format as the given document.*
> *- Please do not put the code itself. Keep the original format.*
> *- Keep the same sentences as much as possible if the outdated document exists.*

2) Updating outdated document snippets: Finally, in the second step, the prompt is given to the LLM to get the updated API document snippet. The input is the prompt, constructed by Step C.1), and the output is the updated API document snippet created by the LLM.

IV. Experimental Setup

A. Research questions

To explore whether LLMs can effectively update API documentation, we identify five research questions (RQs), as follows:

TABLE I: Project details, including the most recent version at the time of the experiment, total lines of source code, number of releases based on GitHub repository tags, and the total number of contributors.

Project	Version	#Lines	#Releases	#Contributors
Elasticsearch Java	v8.17.1	401,906	102	33
Spring-boot	v3.5.0-M1	459,736	336	1,133
Hibernate-orm	7.0.0.Beta3	1,338,351	366	562
WildFly	35.0.0.Final	570,025	163	368
Vert.x	5.0.0.CR3	141,033	182	266

TABLE II: Number of API changes for each project, including modifications, deletions, additions, and the total number of changes.

Project	#Modifications	#Deletions	#Additions	#Totals
Elasticsearch Java	2	1	7	10
Spring-boot	6	2	6	14
Hibernate-orm	5	4	1	10
WildFly	10	2	0	12
Vert.x	5	4	2	11
Totals	28	13	16	57

- RQ1. How accurately do LLMs identify and update API references in outdated API documents?
- RQ2. How accurately do LLMs identify and update API references *per change type*?
- RQ3. How closely do LLMs updates of the API documents match the ground truth updates?
- RQ4. How closely do the LLM updates of the API documents match the ground truth updates *per change type*?
- RQ5. To what extent do developers accept the LLM updates?

B. Target Projects

We selected five target projects from open-source repositories on GitHub. The selection criteria required that the project was written in Java and used the Asciidoctor tool[4] for formatting human-written API documentation. To identify suitable projects, we examined the most popular repositories and checked whether they contained API documentation in $.asciidoc$ or $.adoc$ files. This selection process resulted in the five projects shown in Table I.

As shown in Table I, the projects span many different sizes, ranging from 141,033 lines of code to 1,338,351. For reference, the table also includes the most recent version of the project at the time of the experiment.

C. Large Language Models

In the experiment, we used five large-scale language models (LLMs): GPT-4o mini, Claude, Gemini, Llama 3, and Deepseek. We used the API services of GPT-4o mini, Claude 3.7 Sonnet, and Gemini 2.0 Flash, respectively. For the Llama 3 and Deepseek models, we leveraged llama3:70b and deepseek-r1:70b with Ollama[5] in a local environment.

D. Ground-truth Data

We use the developers' manual updates as ground-truth data. To identify these updates, the first author of this paper started with the latest version of each project and examined the changes in API documentation between two consecutive versions. To reduce the time required for a manual investigation, a semi-automated approach was employed. This process involved breaking down the API document down to the document snippet level, as described in Step III.B.1. The author then identified the snippets that contained API references and manually compared code-relevant updates between corresponding snippets from successive revisions. The investigation continued until 10 ~ 15 ground-truth samples were collected for each project. I total, the process took two weeks for the five projects.

Across all projects, 57 ground truth updates were identified. Table II shows the detailed information for each project and each change type. For example, for Elasticsearch Java, there were 10 samples in the ground-truths dataset: 2 modifications, 1 deletion, and 7 additions.

E. Measurements

First, to measure the accuracy of LLM's identification of outdated API references, we used Precision Recall, and F1-scores, as shown in equation (1) and (2). Precision measures the proportion of correctly identified updates among all updates made by the LLM, while recall quantifies the proportion of correctly identified updates relative to all ground-truth. The F1 score, defined as the harmonic mean of precision and recall, provides a balanced measure of the performance.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{1}$$

where TP (True Positives) represents the number of updates correctly identified by the LLM, FP (False Positives) refers to updates made by the LLM that are not present in the ground-truth dataset, and FN (False Negatives) denotes updates present in the ground-truth dataset but missed by the LLM.

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2}$$

Second, to understand how closely an LLM updates API document snippets to a human, we used three metrics BLEU (Bilingual Evaluation Understudy), Rouge-L (Recall-Oriented Understudy for Gisting Evaluation-L), and Meteor (Metric for Evaluation of Translation with Explicit ORdering).

The BLEU (Bilingual Evaluation Understudy) metric evaluates the quality of generated text by comparing it to human written text. Equation (3) calculates BLEU by multiplying the Brevity Penalty (BP) with the Geometric Average of the Precision Scores.

[4]The Asciidoctor tool assists developers to format API documentation and parse it into various formats. Refer to https://asciidoctor.org/.

[5]https://ollama.com/

$$\text{BLEU} = \text{BP} \cdot \exp\left(\frac{1}{N}\sum_{n=1}^{N}\log p_n\right) \quad (3)$$

Equation (4) calculates Brevity Penalty (BP), where $|c|$ represents the length of the generated text by an LLM and $|r|$ represents the length of the human-written text.

$$\text{BP} = \begin{cases} 1, & \text{if } |c| > |r| \\ \exp(1 - \frac{|r|}{|c|}), & \text{if } |c| \leq |r| \end{cases} \quad (4)$$

Equation (5) calculates the precision score of n-grams. In the equation, $Count_{clip}(n)$ represents the number of clipped, correctly predicted n-grams between the generated text and the human-written text. $Count(n)$ is the total number of n-grams in the generated text, and N represents the maximum degree of the n-grams. In our experiment N is 4.

$$p_n = \frac{\text{Count}_{\text{clip}}(n)}{\text{Count}(n)} \quad (5)$$

ROUGE-L (Recall-Oriented Understudy for Gisting Evaluation-L) evaluates the similarity between a generated text and a human-written text based on the longest common subsequence (LCS). Equation (6) calculates the Precision, Recall, and F1-Score for ROUGE-L. In the equation, $LCS(c, r)$ denotes the length of the longest common subsequence of two sentences c and r. In this paper, LOUGE-L refers to the F1-Score in equation (6).

$$P = \frac{LCS(c,r)}{|c|}, \quad R = \frac{LCS(c,r)}{|r|}, \quad F1 = \frac{2 \cdot P \cdot R}{P + R} \quad (6)$$

METEOR (Metric for Evaluation of Translation with Explicit ORdering) extends BLEU by considering synonyms, stemming (root forms of words), and word order [25]. It uses exact word matches first, then checks for synonyms and paraphrases. Equation (7) calculates the METEOR metric by taking into account the harmonic mean, F_m, in Equation (8) and the chunk penalty in Equation (9).

$$\text{METEOR} = F_m \cdot (1 - Penalty) \quad (7)$$

Equation (8) finds the number of matched unigrams between generated text and human-written text. First, it counts the number of unigrams that match exactly, and then it counts the unigrams that match with synonyms. Based on the matched unigrams, Equation (8) calculates the harmonic mean, F_m.

$$P = \frac{match(c,r)}{|c|}, \quad R = \frac{match(c,r)}{|r|}, \quad F_m = \frac{10 \cdot P \cdot R}{9P + R} \quad (8)$$

Equation (9) calculates the chunk penalty, where chunks are a set of consecutive aligned words between the human-written text and the generated text. The more similar the two texts are, the fewer chunks there are and the smaller the penalty.

$$Penalty = 0.5 \cdot \left(\frac{|chunks|}{match(c,r)}\right)^3 \quad (9)$$

F. Experimental procedures

We describe the experimental procedures for RQs. For RQ1, RQ2, RQ3, and RQ4, we conducted an experiment, based on the ground-truth dataset described in Section IV-B. To prepare the experiment, we first applied the proposed method to the API document snippets in the ground-truth dataset and obtained the updated document snippets per each project. Then, we compared the document snippets updated by an LLM with the document snippets updated by developers in the ground-truth dataset.

1) Experimental procedure for RQ1: For RQ1, we evaluated if API references in outdated API document snippets were accurately updated. An API reference was marked as a true positive if it was correctly updated to match the changes made by developers in the ground-truth dataset. If it remained unchanged despite requiring an update, it was marked as a false negative. We then calculated the precision, recall, and F1 score according to Equations (1) and (2).

2) Experimental procedure for RQ2: For RQ2, we analyzed the F1 scores that we obtained from the experiment. To compare the accuracy across different change types, we grouped F1 scores into three groups: Modifications, Deletions, and Additions.

3) Experimental procedure for RQ3: For RQ3, we took the API document snippets updated by an LLM and compared them to the API document snippets updated by developers in the ground-truth dataset. To assess their similarity, we used three metrics: BLUE, ROUGE-l, and METEOR.

4) Experimental procedure for RQ4: For RQ4, we analyzed the METEOR scores from RQ3 and compared across change types. To accomplish this, we grouped the scores into Modifications, Deletions, and Additions.

5) Experimental procedure for RQ5: For RQ5, we applied the proposed approach to the latest version of each project's API documentation. We manually examined the API snippets updated by the LLM, reviewing them sentence by sentence to ensure accuracy. Finally, we reported the identified API document updates as issue reports for developer review.

V. Experimental Results

A. Results for RQ1

To answer RQ1, we show the average F1-score of all projects in both Figure 2 and Table 3. From these results, we observe that GPT-4o and Claude maintain a consistently higher F1-score than Deepseek, Llama3, and Gemini over projects. Both Claude and GPT-4o maintain scores above or equal to 0.9. Deepseek performs only moderately worse at around 0.8. LLama3 performs at 0.735. Gemini performs at 0.697.

Summary: Given change rule information, GPT-4o showed the highest averaged F1-score of 0.921, and Claude showed the second highest F1-score of 0.900 in identifying and updating API references in API document snippets.

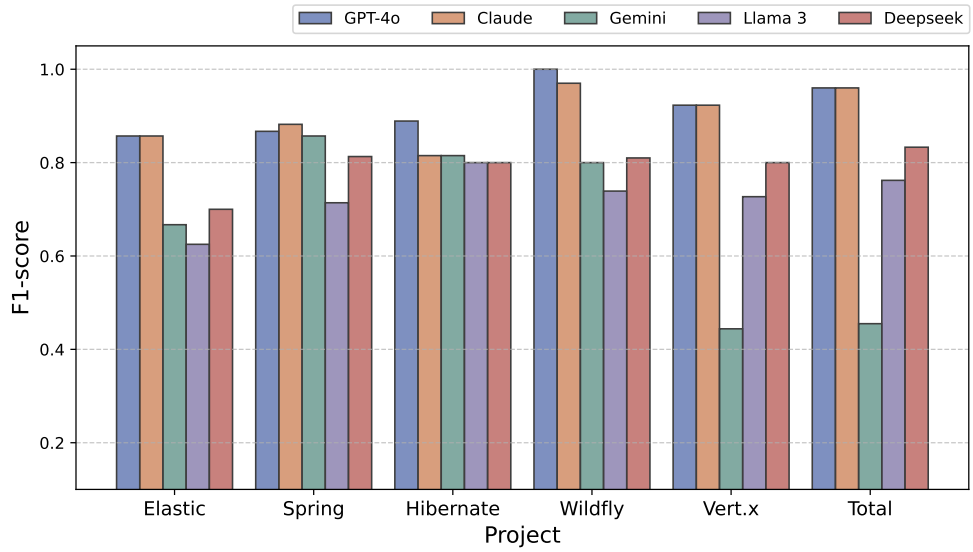

Fig. 2: Average F1-score of LLMs in identifying and updating API references in API document snippets across projects.

TABLE III: F1-scores of LLMs in identifying and updating API references in API document snippets.

	GPT-4o	Claude	Gemini	Llama 3	Deepseek
Elastic-search	**0.857**	**0.857**	0.667	0.625	0.700
Spring-boot	0.867	**0.882**	0.857	0.714	0.813
Hibernate-orm	**0.889**	0.815	0.815	0.800	0.800
Wildfly	**1.000**	0.970	0.800	0.739	0.810
Vertx	**0.960**	**0.960**	0.455	0.762	.833
Total	**0.921**	0.900	0.697	0.735	0.797

B. Results for RQ2

For RQ2, we analyzed the F1 scores according to change types. Figure 3 shows a box plot of F1 scores per each change type. The median value for the modification case is close to 1, with low variability and consistent performance. The median value for the deletion case is also close to 1 and has a similar distribution to that of the modification case. F1 scores for the addition case are more volatile, ranging from 0 to 1.

Based on our analysis, we found that, for modification and deletion cases, LLMs successfully updated document snippets. However, they often removed pieces of code information that should have been retained (FPs). For the case of additions, the LLMs often missed pieces of code information that should have been added (FNs) and frequently inserted source code that was not tagged in the AsciiDoc format (FPs).

Summary: LLMs accurately updated document snippets for modification and deletion cases. The performance of LLMs for addition case was varied, owing to frequently missed information or incorrect references to code.

C. Results for RQ3

For RQ3, we evaluated the similarity between LLM document updates and the ground truth updates. While the previous

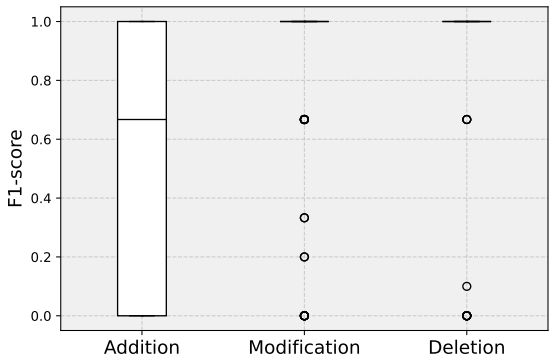

Fig. 3: F1-scores according to the change types.

RQs focused on the factual correctness of the updates, this RQ examines how closely the updates resemble the manual versions. Table IV shows milarity scores using the BLUE, ROUGE-l, and METEOR metrics. The average scores in the last row of each metric indicate that GPT-4o consistently achieved the highest values.

Looking in detail, GPT-4o scores the highest across 7 out of 15. In the 8 cases where it is not the top performer, it ranks second in 7 of them. In all but one case, GPT-4o receives higher similarity scores to the updated documentation than the original, outdated documentation. This indicates that GPT-4o's updates bring the documentation closer to the ground-truth version created by developers in most cases.

The one exception is the WildFly project using the METEOR metric, where the outdated document snippets show the highest similarity to the updated documentation, suggesting minimal changes were made between versions. In such cases, the LLM may be introducing unnecessary modifications, leading to lower similarity scores.

TABLE IV: The similarity between LLM updates and the ground truths

Metrics	*Projects*	*Outdated*	*GPT-4o*	*Claude*	*Gemini*	*Llama 3*	*Deepseek*
BLEU	Elasticsearch Java	0.275	0.448	**0.454**	0.428	0.345	0.336
	Spring-boot	0.426	0.488	**0.499**	0.417	0.480	0.412
	Hibernate-orm	0.707	0.759	**0.761**	0.749	0.626	0.616
	WildFly	0.879	**0.881**	0.866	0.771	0.752	0.764
	Vertx	0.788	**0.804**	0.763	0.716	0.751	0.743
	Total	0.614	**0.672**	0.665	0.610	0.591	0.572
ROUGE-L	Elasticsearch Java	0.297	0.421	**0.429**	0.428	0.387	0.357
	Spring-boot	0.491	0.582	**0.604**	0.540	0.570	0.520
	Hibernate-orm	0.799	0.836	**0.837**	0.831	0.706	0.708
	WildFly	0.909	**0.923**	0.922	0.870	0.857	0.849
	Vertx	0.843	**0.858**	0.812	0.825	0.806	0.804
	Total	0.667	**0.723**	0.721	0.696	0.668	0.648
METEOR	Elasticsearch Java	0.326	**0.637**	0.599	0.627	0.478	0.504
	Spring-boot	0.470	0.606	**0.620**	0.555	0.597	0.526
	Hibernate-orm	0.836	**0.861**	0.836	0.834	0.765	0.743
	WildFly	**0.942**	0.936	0.922	0.833	0.851	0.825
	Vertx	0.820	**0.839**	0.832	0.795	0.811	0.796
	Total	0.676	**0.771**	0.759	0.722	0.700	0.675

Based on the METEOR score, we performed the Wilcoxon Signed-Rank test to verify statistical significance. GPT-4o and Claude were found to be significantly different from outdated API documentation ($p < 0.05$). On the other hand, Gemini, Llama 3, and Deepseek were found to be statistically insignificant ($p \geq 0.05$).

Summary: Based on ground truth updates, GPT-4o yielded consistently highest values. This suggests that GPT-4o's updates bring the documentation closer to the ground-truth version created by developers in most cases.

D. Results for RQ4

For RQ4, we evaluated the similarity scores from RQ3, comparing them across the different the change types. For comparison, we chose to focus on the METEOR metric. Figure 4 shows the METEOR scores for additions, deletions, and modifications.

In the additions case, GPT-4o, Claude, and Gemini all performed well. However, the interquartile range (IQR) of GPT-4o is narrower compared to Claude and Gemini, indicating that GPT-4o produced more consistent METEOR values than the other two models. In all cases, the LLM-generated updates received higher scores than the outdated documentation.

In the modification case, GPT-4o received the highest average METEOR score because it successfully updated only the API references. In contrast, other models made additional updates to other parts of the documentation, which were not necessary. As a result, the METEOR value of GPT-4o is slightly higher than that of the outdated API document snippets, while the METEOR scores for the other models are lower than those of the outdated documentation.

In the deletion case, Claude showed the highest averaged METEOR value. However, GPT-4o, Claude, and Gemini all received higher average METEOR scores, than the average value for the outdated API document snippets.

Summary: While Claude showed the highest METEOR values in the deletion case, GPT-4o showed consistently high METEOR values in three cases.

E. Results for RQ5

To understand if developers accept the LLM updates, we shared the issue reports from the LLMs updates on the most recent version of the project and received developers' feedback on five cases.

There are two lessons learned. First, the updating suggestions should be concrete. For the hibernate-orm project, we reported three outdated API references with update suggestions at sentence level [6]. According to the developer's request, the first author created a pull request with the suggested edits [7]. In the project, two concrete update suggestions were accepted by developers. However, one update suggestion was not, because the developers asked "Which configuration settings?" The API name $GenerationTime$ related to the third suggestion belongs to the deletion case in our update, and we have not found a replacement for it. However, the developer expected to describe a replacement rather than delete the code information.

Second, accurate outdated information is important in such a report. For the spring-boot project, we reported two outdated API references with update suggestions at sentence level [8]. Unfortunately, two outdated API references were not accepted by developers. Then, developers mentioned, "We're a small team, please be considerate of our time before submitting the output of an automated tool." The reason is that our initial Java parser did not properly analyze some source files due to an unmatched Java version. We inaccurately reported outdated API references. This is not directly related to the updates by LLM, which are the main focus of this study, but it does mean

[6]https://hibernate.atlassian.net/browse/HHH-18993
[7]https://github.com/hibernate/hibernate-orm/pull/9521
[8]https://github.com/spring-projects/spring-boot/issues/43645

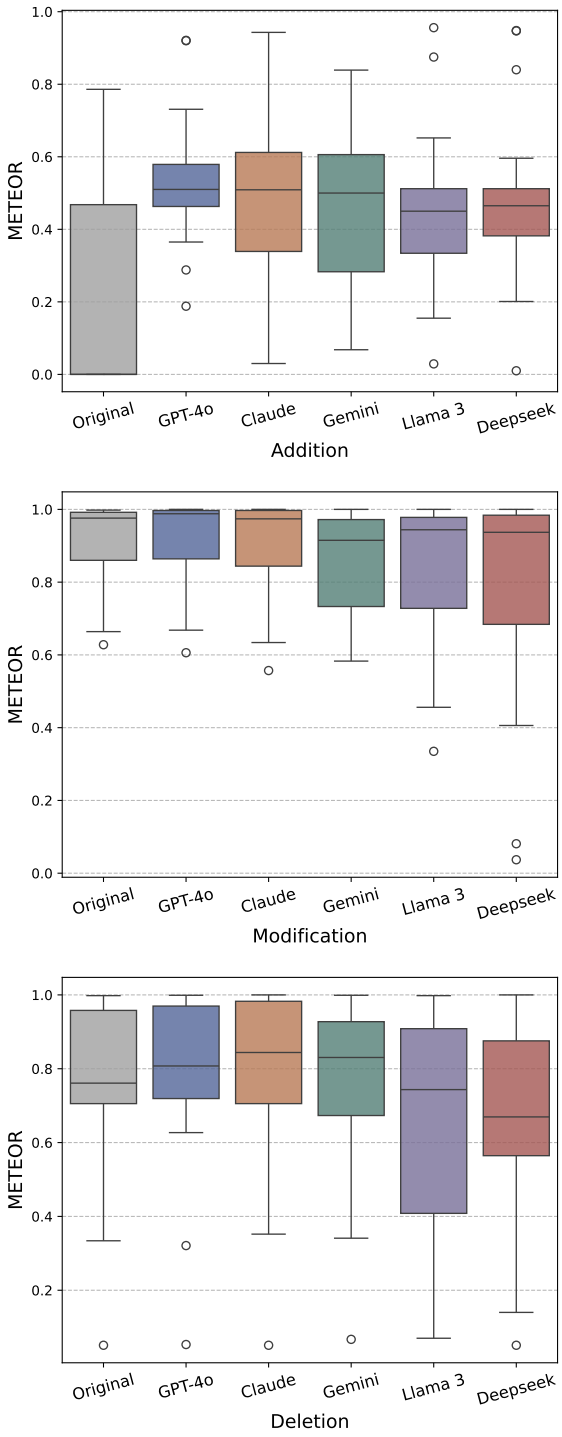

Fig. 4: The similarity between LLM document updates and the ground truth updates according to the change types.

that the API change information must be accurately extracted in the first phase of our approach to be effective.

Summary: Developers accepted two out of five updating suggestions; they asked us to create a pull request by ourselves to reflect the changes.

VI. Discussion

Our experimental results demonstrated higher F1-Scores, BLEU values, and METEOR values of GPT-4o than those of the outdated API document snippets. This suggests that GPT-4o's updates improved the alignment of the documentation, moving it closer to the intended ground-truth updates. In this section, we demonstrate the examples of GPT-4o's updates and discuss advantages and disadvantages of LLM updates.

GPT-4o's update for a modification case .....
Configure Two DataSources:...
If you create your own 'DataSource', the auto-configuration backs off. In the following example, we provide the _exact_ same feature set as the auto-configuration provides on the primary data source:
-include-code::MyDataSourcesConfiguration[]
+include-code::MyAdditionalDataSourceConfiguration[]

GPT-4o's update for a deletion case .....
==== Indexing application objects ...
["source","java"]

include-tagged::{doc-tests-src}/usage/IndexingBulkTest.java[bulk-objects]

<1> Adds an operation (remember that ¡¡listsandmaps,list properties are additive>>)...
==== Indexing raw JSON data ...
-["source","java"]

-include-tagged::{doc-tests-src}/usage/IndexingBulkTest.java[read-json]

-We can now read the contents of the log directory and send it to es:
-["source","java"]

-include-tagged::{doc-tests-src}/usage/IndexingBulkTest.java[bulk-json]

GPT-4o's update for an addition case .....
+==== Using the Bulk Ingester

+The 'BulkIngester' is a utility for efficiently processing large volumes of bulk operations, such as indexing logs. It allows you to define a batch of operations that are executed in bulk to improve performance. . .
["source","java"]

+include-tagged::{doc-tests-src}/usage/IndexingBulkTest.java[bulk-ingester-setup]

+<1> Configures the 'BulkIngester' with the Elasticsearch client. You must provide an instance of 'esClient' to facilitate communication with the Elasticsearch server. . .

Ground-truth update for an addition case .....
+include::{docs-java}/features/testing/springbootapplications/auto-configuredspringrestdocs/withwebtestclient/MyWebTestClientBuilder-CustomizerConfiguration.java[]
GPT-4o's update .....
+include::{code-examples}/web/client/WebTestClientRestDocumentation-Example.java[tag=customizer]
Claude's update .....
+include::{code-examples}/features/testing/springbootapplications/auto-configuredspringrestdocs/withwebtestclient/MyWebTestClientBuilder-CustomizerConfiguration.java[]
Gemini's update .....

Llama's update .....
+include::{code-examples}/web/client/MyWebTestClientBuilder-CustomizerConfiguration.java[tag=restDocumentation]
Deepseek's update .....
+include::{code-examples}/web/client/MyWebTestClientBuilder-CustomizerConfiguration.java[tag=customizer]

Fig. 5: Examples of document updates

A. Examples of GPT updates

In Figure 5, we demonstrate examples of GPT's updates to the documentation. The example in the first row illustrates a *modification* update to the API document snippets in Spring-boot. In this case, GPT-4o only modified the API name in the snippet without altering the content of the text. Through comparison with the ground-truth updates, we verified that GPT-4o accurately updated the API reference.

In the second, we present a *deletion* update to the API document snippets in Elasticsearch Java. The document section originally had three API references, but the API function related to read-json had been removed from the source code. With this information, GPT-4o deleted the entire document snippet related to read-json to reflect the code change.

The example in the third row illustrates an *addition* update to the Elaticsearch Java Project. In this case, a function related to bulk-ingester-setup was added to the source code. Therefore, the intended updates was for GPT to generate the API document snippet describing bulk-ingester-setup. The example demonstrates that GPT-4o successfully generated an API references describing bulk-ingester-setup, including relevant source code and a description of it in a similar format to the other descriptions.

The example in the last row illustrates an *addition* update to the Spring-boot Project. With the example, we discuss the variations between LLMs in updating a single code reference. In this case, GPT-4o failed to reflect the path, file name, and namespace {docs-java}. GPT-4o also added an unnecessary tag $customizer$ (FP). Next, Claude correctly updated everything except the namespace (TP). Gemini didn't generate the relevant code information at all (FN). Llama3 failed to reflect the path, file name, and namespace and added an invalid tag, $restDocumentation$ (FP). Deepseek also failed to reflect reflect the path, file name, namespace (FP).

B. Advantages and Disadvantages of LLM updates

Through these examples, we gained insight into both the advantages and disadvantages of LLM updates. LLMs can effectively update API document snippets based on change rules, without relying on traditional linguistic analysis. When provided with the relevant API change rules in the prompt, LLMs—particularly GPT-4o—accurately applied this information to update the API document snippets.

On the other hand, we also observed certain drawbacks of the LLM-generated updates. LLMs sometimes made unnecessarily updates to content that was unrelated to the code changes. This explains why outdated API document snippets showed a higher METEOR value for the WildFly project in Table IV. Developers typically update API documentation with minimal sentence modifications, whereas LLMs tend to introduce larger, sometimes unnecessary changes. Also, we observed variation across different LLM models where some LLMs included the source code itself or added a extensive descriptions, deviating from the expected updates.

VII. Threats to Validity

Internal threats to validity are as follows. First, we utilized 57 ground-truths found in the revisions of API documentation. The limited number of ground truths makes it difficult to guarantee the reliability and consistency of the experimental results. Second, our ground-truths were limited to document snippets containing an outdated API reference. However, there were multiple outdated API references in a single document snippet. In future research, it is worth collecting such ground-truths for document updates.

External threats to validity are as follows. We conducted our experiments on the Java projects that use the Asciidoctor tool. There are several generalization issues. First, the phase 2 of our approach assumes the markup language of the asciidoctor tool. In this respect, if other API documentation tools are used, the phase may need to be modified to be reusable. However, this change is limited to modifying the approach in splitting API documentation in phase 2. It does not impact the overall experimental results. Second, our experiment is limited to Java projects. We tried to extend our experiment to Python projects, but soon found that Python projects use different documentation methods and tools. Therefore, our results are limited to the Java projects and extending our study to other languages could be another study.

VIII. Conclusion

In this paper, we propose updating API documentation by augmenting LLMs with code changes. To evaluate this approach, we conduct an experiment using API document snippets from 5 different open-source projects. In the evaluation results, GPT4o and Claude correctly identify and update code information in outdated API document snippets, achieving F1-scores of 0.921 and 0.900, respectively. GPT-4o's updates also aligned with the manual updates, achieving a METEOR score of 0.771, a ROUGE-l score of 0.723, and a BLEU score of 0.672 - higher than those of the outdated API sections. This indicates that the document snippets updated by GPT-4o are more similar to the documents updated by humans. Claude also demonstrated comparable performance in updating API document snippets. Our results demonstrate the potential of LLMs in updating API documentation using change rules.

We consider the following future work. First, we did not evaluate the process of combining updated document snippets to create a complete API documentation. Therefore, we plan to automate the full process by linking snippets to produce a fully updated API documentation. Second, our study focused solely on updating API documents. In the future, we aim to extend this research to updated multiple software artifacts throughout the software development process. We will also continue to conduct user studies and apply our approach to real-world projects to further validate its effectiveness.

References

[1] S. C. B. De Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, pp. 68–75, 2005.

[2] S. Lee, R. Wu, S.-C. Cheung, and S. Kang, "Automatic detection and update suggestion for outdated api names in documentation," *IEEE Transactions on Software Engineering*, vol. 47, no. 4, pp. 653–675, 2019.

[3] Oracle, "How to write doc comments for the javadoc tool," 2024. Accessed: 2025-03-05.

[4] D. van Heesch, "Documenting the code," 2024. Accessed: 2025-03-05.

[5] H. Zhong and Z. Su, "Detecting api documentation errors," in *Proceedings of the ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pp. 803–816, 2013.

[6] W. S. Tan, M. Wagner, and C. Treude, "Detecting outdated code element references in software repository documentation," *Empirical Software Engineering*, vol. 29, no. 1, p. 5, 2024.

[7] W. S. Tan, M. Wagner, and C. Treude, "Wait, wasn't that code here before? detecting outdated software documentation," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 553–557, IEEE, 2023.

[8] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza, "Software documentation issues unveiled," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 1199–1210, IEEE, 2019.

[9] B. Dagenais and M. P. Robillard, "Using traceability links to recommend adaptive changes for documentation evolution," *IEEE Transactions on Software Engineering*, vol. 40, no. 11, pp. 1126–1146, 2014.

[10] C. Yang, J. Liu, B. Xu, C. Treude, Y. Lyu, M. Li, and D. Lo, "Apidocbooster: An extract-then-abstract framework leveraging large language models for augmenting api documentation," *arXiv preprint arXiv:2312.10934*, 2023.

[11] N. Jain, R. Kwiatkowski, B. Ray, M. K. Ramanathan, and V. Kumar, "On mitigating code llm hallucinations with api documentation," *arXiv preprint arXiv:2407.09726*, 2024.

[12] L. Fan, J. Liu, Z. Liu, D. Lo, X. Xia, and S. Li, "Exploring the capabilities of llms for code change related tasks," *ACM Transactions on Software Engineering and Methodology*, 2024.

[13] K. R. Dearstyne, A. D. Rodriguez, and J. Cleland-Huang, "Supporting Software Maintenance with Dynamically Generated Document Hierarchies," Aug. 2024. arXiv:2408.05829 [cs].

[14] J. Y. Khan and G. Uddin, "Automatic code documentation generation using gpt-3," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–6, 2022.

[15] Y. Su, C. Wan, U. Sethi, S. Lu, M. Musuvathi, and S. Nath, "Hotgpt: How to make software documentation more useful with a large language model?," in *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pp. 87–93, 2023.

[16] K. Lazar, M. Vetzler, G. Uziel, D. Boaz, E. Goldbraich, D. Amid, and A. Anaby-Tavor, "Specrawler: Generating openapi specifications from api documentation using large language models," *arXiv preprint arXiv:2402.11625*, 2024.

[17] C. Wang, K. Huang, J. Zhang, Y. Feng, L. Zhang, Y. Liu, and X. Peng, "How and why llms use deprecated apis in code completion? an empirical study," *arXiv preprint arXiv:2406.09834*, 2024.

[18] Y. Yu, G. Rong, H. Shen, H. Zhang, D. Shao, M. Wang, Z. Wei, Y. Xu, and J. Wang, "Fine-tuning large language models to improve accuracy and comprehensibility of automated code review," *ACM Transactions on Software Engineering and Methodology*, 2024.

[19] A. Imani, I. Ahmed, and M. Moshirpour, "Context conquers parameters: Outperforming proprietary llm in commit message generation," *arXiv preprint arXiv:2408.02502*, 2024.

[20] L. Zhang, H. Zhang, C. Wang, and P. Liang, "Rag-enhanced commit message generation," *arXiv preprint arXiv:2406.05514*, 2024.

[21] J. Li, D. Faragó, C. Petrov, and I. Ahmed, "Only diff is not enough: Generating commit messages leveraging reasoning and action of large language model," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 745–766, 2024.

[22] Y. Zhang, "Detecting code comment inconsistencies using llm and program analysis," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pp. 683–685, 2024.

[23] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th international conference on software engineering*, pp. 483–494, 2018.

[24] D. Silva, J. P. da Silva, G. Santos, R. Terra, and M. T. Valente, "Refdiff 2.0: A multi-language refactoring detection tool," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2786–2802, 2020.

[25] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments," in *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pp. 65–72, 2005.