Supporting Automated Documentation Updates in Continuous Software Development with Large Language Models

Henok Birru, Antonio Cicchetti[a] and Malvina Latifaj[b]
Mälardalen University, Västerås, Sweden
hbu23001@student.mdu.se, {antonio.cicchetti, malvina.latifaj}@mdu.se

Keywords: Continuous Software Development, Continuous Documentation, Large Language Models, Retrieval Augmented Generation.

Abstract: Nowadays software is ubiquitous in our society, making its role increasingly mission critical. Software applications need to be continuously maintained and evolved to keep up with the pace of market demands and emerging issues. Continuous Software Development (CSD) processes are an effective technological countermeasure to the mentioned evolutionary pressures: practices like DevOps leverage advanced automation mechanisms to streamline the application life-cycle. In this context, while handling the application development and implementation is adequately investigated, managing the continuous refinement of the corresponding documentation is a largely overlooked issue.

Maintaining accurate technical documentation in CSD is challenging and time-consuming because the frequent software changes require continuous updates and such a task is handled manually. Therefore, this work investigates the automation of documentation updates in correspondence with code changes. In particular, we present CodeDocSync, an approach that uses Large Language Models (LLMs) to automate the updating of technical documentation in response to source code changes. The approach is developed to assist technical writers by summarizing code changes, retrieving updated content, and allowing follow-up questions via a chat interface.

The approach has been applied to an industrial scenario and has been evaluated by using a set of well-known predefined metrics: contextual relevancy, answer relevancy, and faithfulness. These evaluations are performed for the retriever and generator components, using different LLMs, embedding models, temperature settings, and *top-k* values. Our solution achieves an average answer relevancy score of approximately 0.86 with OpenAI's *gpt-3.5-turbo* and *text-embedding-3-large*. With an emotion prompting technique, this score increases to 0.94, testifying the viability of automation support for continuous technical documentation updates.

1 INTRODUCTION

The ever increasing ubiquity of software in our everyday lives and the global market competition are exacerbating the evolutionary pressures applications are subject to. In particular, emerging demands and/or issues related to a software application need to be quickly addressed to avoid losing users or even worse risking penalties due to malfunctions. The pressures for frequent updates stress the effectiveness of the adopted development process, since any friction in the workflow tends to cause delays and additional workload.

Continuous software development (CSD) methodologies target specifically scenarios in which applications require frequent updates. In particular, CSD proposes to maximize the automation for repetitive and time-consuming tasks to both save time and also to use skilled personnel for core aspects of the product development (Bosch, 2014; Chui et al., 2016). Among the software development phases an often overlooked part should be devoted to prepare technical documentation (TD); TD is usually released together with the software product and is meant to instruct on how to effectively use the application.

Especially in industrial contexts, it is critical to keep the documentation up-to-date; in this respect, even though some approaches to manage fast-paced release processes exist (e.g. Documentation as Code (doc-as-code)), this task remains largely manual. Notably, documentation writers (also known as technical writers) might need to rearrange the contents based

[a] https://orcid.org/0000-0003-0416-1787
[b] https://orcid.org/0000-0002-2754-9568

Birru, H., Cicchetti, A. and Latifaj, M.
Supporting Automated Documentation Updates in Continuous Software Development with Large Language Models.
DOI: 10.5220/0013286800003928
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 20th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2025)*, pages 92-106
ISBN: 978-989-758-742-9; ISSN: 2184-4895
Proceedings Copyright © 2025 by SCITEPRESS – Science and Technology Publications, Lda.

on the code changes, update code snippets or screenshots, and other similar documentation refinements that require advanced understanding of change consequences (Khan and Uddin, 2023).

Large Language Models (LLMs) are increasingly supporting and even completely replacing labor-intensive tasks in software development processes. Code generation (Austin et al., 2021; Chen et al., 2021), code summarization (Ahmed and Devanbu, 2022), and code review (Lu et al., 2023) are just few examples of powerful features that come in handy, especially in large and rapidly changing scenarios. Not surprisingly, the application of LLMs has also been investigated for handling software documentation (Su et al., 2023; Khan and Uddin, 2022). Nonetheless, the current solutions do not provide support for handling the update of technical documentation due to source code changes, a very common scenario in CSD that makes such feature essential.

This paper proposes a methodology to set up an end-to-end system, making it possible for technical writers to quickly update technical documentation in response to code changes. The proposed solution uses LLMs and appropriate prompting techniques to provide the following three complementary components: an automatic updater of the existing technical documentation based on source code changes; a change descriptor summarizing how the modifications have been interpreted; a chat replying to custom questions posed by technical writers. The underlying aim is to drastically reduce the effort required to manage the documentation by technical writers and developers.

The proposed methodology is validated by instantiating it within an industrial CSD scenario. The developed code is used as a basic framework on top of which it is possible to create networking related tools. In this respect, features are continuously added and updated, making frequent documentation update a critical requirement. The solution resulting from the instantiation of our methodology is evaluated using predefined metrics achieving an average answer relevancy of 0.86 using OpenAI's *gpt-3.5-turbo* LLM and *text-embedding-3-large* embedding model. Moreover, the score reaches to 0.94 when emotion prompting technique is adopted in the query prompt.

The paper is structured as follows. Section 2 provides background information on the key concepts. Section 3 describes the related work to this research. Section 4 presents the proposed approach, while Section 5 describes the instantiation of the approach in an industrial case study and evaluation results. Section 6 concludes the paper and describes future research directions.

2 BACKGROUND

This section provides basic information about the technical solutions that underpin the proposed solution, notably LLMs and prompt engineering. Moreover, it highlights the needs for continuous documentation and existing aids to clarify the motivation behind this research work.

2.1 Large Language Models

In machine learning (ML), language models are solutions tailored to predict the next word in a sentence based on the rest of that is given as input. The prediction is computed through a probability distribution of the available vocabularies and an estimation of the likelihood of each word in a given context. The technical realization of language models has been evolving over time[1]; the latest ones are called Generative Pre-trained Transformer (GPT)-series LLMs and are the base for the approach proposed in this paper.

LLMs represented a major breakthrough in natural language processing (NLP) research thanks to the adoption of the transformer architecture and the attention mechanism (Vaswani et al., 2017). The discussion of the technical details of LLMs from an AI/ML perspective goes beyond the scope of this work. In our scope, it is worth mentioning that the architecture and the attention mechanism allow parallel computation and the capture of long-distance sequences. These features boost the precision of predictions and keep the computation time relatively tractable; moreover, they allow handling huge amounts of training data (in the order of billions of parameters). Among the usages for which LLMs have shown interesting performances, it is worth mentioning the following automated software engineering tasks (ASET)(Shin et al., 2023):

- **Code Generation:** generate code that fulfills a given natural language description (requirement);
- **Code Summarization:** automatically create coherent, precise, and valuable code comments to assist developers in understanding a chunk of code given as input;
- **Code Translation:** convert code from one programming language to another while keeping the functionality intact;
- **Code Review and Evaluation:** perform static analysis to identify potential faults in a program.

[1]https://www.appypie.com/blog/evolution-of-language-models

In this work, we use LLMs to analyze code changes originating from merge requests and update technical documentation accordingly.

2.2 Prompt Engineering

The input submitted to LLMs is called a prompt and has demonstrated to play a fundamental role for the quality of the generated outputs (Shin et al., 2023). In fact, since LLMs basic goal is predicting words in natural language sentences, generating domain-specific text (like for ASET) might require to direct the LLM towards the correct interpretation of the input and avoid erroneous outputs. Notably, there exists the very well-known problem of model hallucinations, i.e. scenarios in which the LLMs generate nonsense and/or unreliable results (Ji et al., 2023). In fact, a discipline has recently emerged that targets the processes of designing, building, and refining prompts to obtain the most relevant response from an LLM, called Prompt Engineering. (Sahoo et al., 2024).

A selection of prompt engineering techniques includes **Zero-Shot Prompting**, which uses only instructions without examples, and **Few-Shot Prompting**, which improves responses by providing examples in the prompt (Gao et al., 2023). **Chain-of-Thought (CoT) Prompting** introduces intermediate reasoning steps for handling complex tasks and generating explanations, while **Self-Consistency Prompting** extends CoT by exploring multiple reasoning paths and selecting the most consistent outcome (Wang et al., 2022). **Prompt Chaining** breaks tasks into sub-tasks with detailed sub-prompts, aiding in complex problem-solving. **Tree of Thoughts (ToT) Prompting** explores multiple generation paths, evaluating and selecting the best option (Yao et al., 2023). **Expert Prompting** improves LLM's output by creating detailed, domain-specific prompts that guide the model to respond as a distinguished expert in the relevant field (Xu et al., 2023a). **Emotion Prompting** incorporates emotional stimuli into prompts, improving LLMs' performance across various tasks (Li et al., 2023). Finally, **Retrieval Augmented Generation (RAG)** incorporates information retrieval to provide domain-specific knowledge, ensuring consistency and reducing hallucinations (Sahoo et al., 2024).

As discussed in more detail in Section 5, our approach proposes to use RAG as a prompting technique.

2.3 Continuous Software Development and Documentation

A well-maintained documentation is of paramount importance, especially to support practitioners effectiveness and keep an adequate level of users' experience (Aghajani et al., 2020). This requirement is even more relevant in the case of technical software documentation, that is, the documentation associated with the source code. For instance, software development kit (SDK) and API documentation are examples of documentation that are used to share information among upstream and downstream developers. In this respect, the accuracy of such information allows for the proper construction of derived applications and effective maintenance of the code itself (Rai et al., 2022).

The evolutionary pressures modern software is subject to have pushed development processes towards enhancing speed. In particular, methodologies like Agile and DevOps aim to compress the time required for tasks that do not add tangible value for the user while improving the responsiveness of the process to the need for changes (Bosch, 2014). These trends apparently collide with an adequate maintenance of the documentation, since this task is often seen as a less valuable part of a software product release and can rapidly become a bottleneck when the rate of changes increases. Aghajani et al. (Aghajani et al., 2020) highlight poor documentation related challenges as reported by practitioners, like inaccuracies in information content, erroneous code examples, incorrect comments, and lack of completeness. Additionally, outdated documentation is a commonly reported issue negatively affecting the developer experience (Aghajani et al., 2019). In this respect, automation can play a crucial role in alleviating these issues by enhancing the technical documentation process.

The remainder of this section illustrates some countermeasures introduced with the purpose of supporting continuous technical documentation maintenance and positions the contribution of this paper.

2.4 Documentation as Code

Docs-as-code proposes to adopt a documentation maintenance approach that goes hand-in-hand with the development of the software; software development and documentation shall use the same procedures and tools, including issue trackers, version control systems, lightweight markup languages, automated testing, and incorporating documentation reviews into the writing workflow. The underlying goal is to promote the collaboration between developers and technical writers in effectively producing technical documentation (Holscher, 2024).

Although most of the stages related to documentation writing would benefit from the same automation mechanisms existing for code development, few tasks require additional efforts from the developers and technical writers. Notably, whenever new features are added or large code changes occur, information exchanges are required to clarify the purposes of the new software. In this respect, either developers write the first draft of the documentation based on the code change they have made, or technical writers need to learn more about the technical details of the changes before starting the actual writing.

This work explores the capability of LLMs to analyze developers' code changes and then modify the existing documentation accordingly. The solution aims to replace the efforts required by developers in writing the draft documentation content for every code change in the docs-as-code approach. In addition, it limits the efforts of technical writers to learn and understand the changes the source code has undergone. This support allows documentation maintenance to maintain the speed of source code development and frees precious resources for other core tasks.

2.5 Code Summarization

Code summarization refers to the automated creation of some form of source code documentation. In particular, the summarization can be extractive or abstractive: the former extracts information directly from the code (e.g., the name and type of parameters passed to a function); the latter provides a more advanced description of the code that is not limited to the mere direct reading of the code (Mastropaolo et al., 2023). As expected, many generative models including LLMs produce abstractive summarization, typically natural language summaries of code snippets that can be used as inline comments or integrated into technical documentation along with other descriptions.

Code summarization can be performed at different granularity levels, ranging from line-by-line to a code module; correspondingly, for each granularity level the summarization output will produce different insights about the code, ranging from, e.g., the input and output of a function to the features and interfaces with other modules, respectively (Zheng et al., 2023).

This work investigates the use of summarization techniques to analyze source code changes, and by taking into account existing documentation automatically generate appropriate updates. The changes are given in terms of merge requests, and their impact is determined by examining the entities involved in the changes and their interdependencies with the existing code (and hence documentation). Moreover, the summarization is used to provide the users with a human-friendly description of the changes and leveraged by a chatbot answering technically detailed questions about the modifications.

3 RELATED WORK

Automated documentation has been investigated at length, even before the advent of AI/ML techniques: for example, the Automatic Summary Comment Generator (Sridhara et al., 2010) and Automatic Code Summarization (Moreno et al., 2013) are solutions that leverage heuristics and/or predefined rules to extract and summarize source code information. Moreover, there exist solutions using text retrieval techniques to derive documentation from the structure and syntactical characteristics of the code(Haiduc et al., 2010a; Haiduc et al., 2010b). Although potentially effective, these techniques are less resilient to the evolution of the codebases, since heuristics and rules need to be kept up to date with the evolution history of the source code.

AI/ML first and the advent of LLMs recently have triggered a bursting interest for solutions aiming at the automation of repetitive tasks in the software development process (Batarseh et al., 2021; Nguyen-Duc et al., 2023), including the handling of software documentation (Khan and Uddin, 2022). In general, these solutions can be distinguished between learning-based and pre-trained: the former ones require a preliminary phase in which a typically large dataset is used to build (i.e. train) the model supporting the ML solution (Hu et al., 2018; Moore et al., 2019; Ahmad et al., 2020; Rai et al., 2022; Feng et al., 2020); the latter ones rely on large pre-made models that have demonstrated advanced capabilities in natural language processing, including code generation and summarization (Phan et al., 2021; Ahmad et al., 2021; Husain et al., 2019; Khan and Uddin, 2022). The learning-based solutions are intrinsically more effective when working on the development scenarios they have been trained for. However, they tend to degrade when the codebase is affected by extensive changes (both from a size and a time perspective). On the contrary, pre-trained solutions might show lower effectiveness on specific scenarios, but have the advantage of providing more stable performances in the general case.

With the recent boost of generative AI models there has been an increasing interest in using pre-trained transformer models to automate documentation tasks: models like Cotext (Phan et al., 2021) and PLBART (Ahmad et al., 2021) outperformed existing learning-based solutions like CodeBERT (Feng et al., 2020) in code understanding and summarization. In particular, Codex is a GPT-3 pre-trained model specialized towards natural language and programming languages. The evaluation on CodeSearchNet dataset by Khan et al (Khan and Uddin, 2022) shows that Codex surpasses existing techniques with one-shot learning. Moreover, building upon Codex's capabilities for code understanding and summarization, Khan et al. also proposed an approach to generate code examples for documentation purposes (Khan and Uddin, 2023).

While the approaches discussed so far focus on generating documentation or code examples for individual program units such as functions, our approach goes beyond that scope. It aims to address a broader context by examining all code changes within a single merge request to capture the inter-dependencies and implications of each modification. Furthermore, while most existing works utilize basic prompt techniques such as zero-shot or few-shot prompt engineering for documentation automation tasks, which suffice for their scenario, our approach uses advanced prompting techniques such as RAG to enhance the quality of the generated outcomes.

RepoAgent is an open-source framework that uses LLMs to generate, maintain, and update code documentation (Luo et al., 2024). RepoAgent leverages the entire code repository as context to deduce the functional semantics of target code objects. In particular, meta information, Abstract Syntax Tree (AST) analysis, and project tree organization and reference relationships within the code are used to create representations of code repositories. Subsequently, such representations are used to construct prompts devoted to documentation generation tasks. Our approach leverages merge requests as part of the generation prompts instead of AST; as such, using the RAG prompting technique is critical to achieve appropriate results. In this respect, RepoAgent is evaluated by means of human inspection, while we adopt widespread metrics assessing the quality of the generated results. Besides, although the RAG prompt technique is not central to the RepoAgent approach, it is mentioned as a potential future investigation direction. Interestingly, also for RepoAgent it is planned as future investigation a chat feature to support a better understanding of code changes.

An ongoing open-source project called Autodoc, similar to RepoAgent, uses LLMs to automatically generate documentation directly from source code (Alephium, 2023). Autodoc's approach analyzes the entire codebase with an LLM to produce inline documentation that explains the purpose and function of each part of the code, embedding these explanations directly within the source files. This documentation is primarily focused on detailing code functionality at a granular level. In contrast, our tool is designed to update technical documentation maintained in a separate repository following a docs-as-code approach. Instead of analyzing the entire codebase, our tool processes specific MRs in the context of the existing documentation, allowing the LLM to efficiently generate and update documentation based on recent changes. This includes not only code explanations but also broader context, such as usage guidelines and code examples, all of which are important for maintaining comprehensive and user-focused technical documentation.

It is also worth mentioning that significant efforts have been invested in identifying and correcting discrepancies between the source code and corresponding documentation. In particular, approaches have been proposed to enhance the consistency between code and comments (Wen et al., 2019; Liu et al., 2023; Dau et al., 2023), and between software repository documentation and code references (Tan et al., 2024; Tan et al., 2023).

4 APPROACH

Figure 1 provides a high-level overview of our proposed approach. Since the approach assumes technical documentation is developed using the docs-as-code methodology, documentation is treated as part of the codebase and is placed within the same version control system (VCS) as the source code and merge request (MR). The tool connects to the VCS, retrieves open MRs, analyzes the associated code changes, and combines this information with existing documentation to generate context-aware prompts. These prompts are designed to guide the LLM in fulfilling the three core capabilities: automatically updating documentation, summarizing code changes, and providing interactive support for technical writers. The approach description is structured as follows. Section 4.1 covers the preprocessing of input data. Section 4.2 outlines the workflow for automated documentation updates, followed by Section 4.3, which details the summarization feature. Section 4.4 explains the workflow for the chat engine, and Section 4.5 provides an overview of the selected LLMs. Eventually, Section 4.6 illustrates the metrics used to evaluate the quality of both change retrieval and corresponding generation of documentation updates.

4.1 MR Data Cleaning

To achieve accurate and contextually appropriate responses, it is crucial to provide the LLM with clean and filtered data. In many MRs, certain changes – such as third-party package installations in lock files or modifications to unit and integration test files - are not relevant and introduce unnecessary noise. Our approach minimizes storage overhead and enhances the LLM's performance by filtering out non-essential files (A), focusing exclusively on relevant data. In addition, it empowers technical writers to further refine the data-cleaning process, allowing them to exclude any extra files that do not contribute to the documentation.

4.2 Automatic Documentation Update

Our approach employs the RAG technique to automatically update the technical documentation. As presented in Section 2, RAG is an advanced prompt engineering technique that enhances the LLM's output by integrating external knowledge sources with the model's pre-existing training data. Sources can include databases, repositories, or other knowledge bases; in our implementation, the existing technical documentation serves as the external knowledge source, enabling the system to reference existing documentation content before incorporating the changes. As shown in Figure 1, the RAG pipeline consists of three key stages: i) loading, ii) indexing, and iii) querying.

4.2.1 Loading

The first stage of RAG entails retrieving the external knowledge base which in our case is the existing technical documentation from its storage location (C), assuming a docs-as-code methodology is in place. In this setup, technical documentation is stored as markdown files in a Git repository or accessed from local storage. To enhance efficiency, the system identifies the relevant content directory, excluding non-essential files. The technical documentation is loaded using the LlamaIndex *SimpleDirectoryReader* class, which reads each file as a *document* object. The documentation is typically organized into multiple files, each discussing specific features or topics, and related files are grouped under common directories. Rather than dividing the content into very small chunks, we utilize the existing file structure as our chunking strategy to maintain contextual coherence. By default, LlamaIndex splits files into multiple *nodes* with a chunk size of 1024 tokens and a 20-token overlap, resulting in files being segmented across multiple *nodes* for efficient processing and retrieval. The chunking parameters are adjusted to retain context and minimize information loss.

4.2.2 Indexing

The second stage of RAG is indexing, which involves organizing the nodes into a structure optimized for fast retrieval. This process enables efficient access to relevant information based on semantic similarity. Indexes are built from the nodes created during the loading phase, and once indexed, these nodes are fed to an embedding model (D) and converted into vector embeddings stored in the *VectorStoreIndex* (E). Embeddings represent words or phrases as dense vectors of real numbers in a continuous vector space. These vectors are lists of floating point numbers. Unlike the traditional encoding techniques such as one-hot encoding, embeddings encode semantic meaning and relationships between words which is very useful for the LLMs to understand the similarities and differences between words on their context. The embedding process starts by tokenizing the given text. The size of the vector used to represent each token in the embedding space is called the embedding dimension. The dimension size or the number of values required to represent a token is different across various models. Additionally, embedding models have a maximum token size, which refers to the number of tokens the model can process at a time. For texts that have longer size than the capacity of the embedding model, they need to be broken down into smaller chunks before being converted to embeddings.

The embedding process requires an embedding model that transforms text into numerical representations to capture semantic meaning. The choice of embedding model directly impacts both the performance and cost-efficiency of the system. In this approach, we experimented with two different embedding models. The first, *BAAI's bge-small-en-v1.5* [2], is a general-purpose embedding model ranked 37th on the Massive Text Embedding Benchmark (MTEB) Top-40 leaderboard. With 384 embedding dimensions and a token capacity of 512, this model offers a compact structure and is available for free on HuggingFace, making it a cost-effective option. In contrast, *OpenAI's text-embedding-3-large* model ranks 14th on the MTEB leaderboard and offers 3072 embedding dimensions with a significantly higher token limit of 8191. However, this advanced capability incurs additional costs, distinguishing it from the cost-

[2]https://huggingface.co/BAAI/bge-small-en

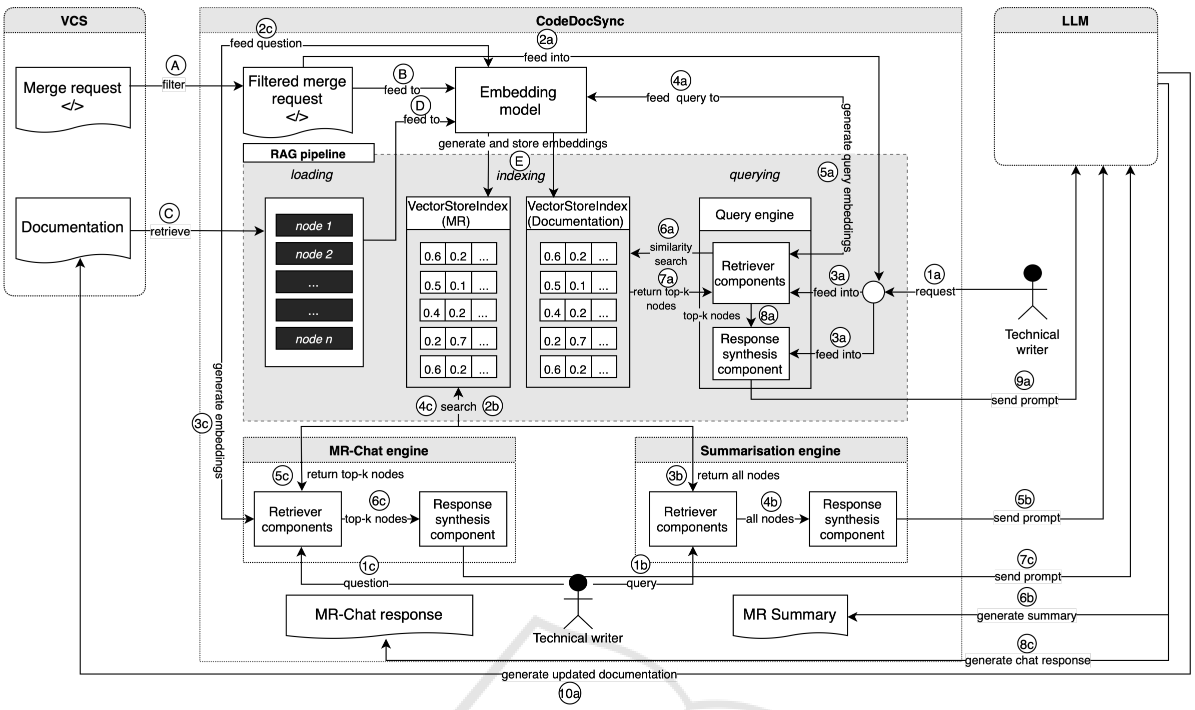


Figure 1: CodeDocSync workflow.

free alternative.

4.2.3 Querying

The last stage of RAG entails the querying process. The technical writer's query (1a) is combined with the code diff from the filtered MR (2a) and additional instruction statements. To prepare the query prompt, the approach employs prompt chaining, emotion prompting, and expert prompting. Prompt chaining directs the model through a series of prompts, proving particularly useful for multi-step tasks (Gadesha and Kavlakoglu, 2024). Prompts are sequentially sent to the LLM using the *create* and *refine* prompt templates. The initial prompt employs the *create* template to generate a first response from the LLM, which is then refined through subsequent prompts using the *refine* template. This process builds upon each response, improving the model's focus and precision across steps until all nodes are processed. Emotion prompting improves LLM's performance by including psychological principles into prompts through emotional stimuli (Li et al., 2023). In our prompt templates, phrases like *"You'd better be sure"* are employed, drawing on the *social identity* theory to improve the quality of responses from the model. Expert prompting improves LLM's responses by positioning it as an authoritative figure. Starting prompts with phrases like *"As a technical writer specializing in developer documentation, your task is to update the outdated part of the documentation"*, we guide the model to respond with the expertise needed for technical documentation updates (Xu et al., 2023b).

The query prompt is initially processed by the *VectorIndexRetriever* (3a), which employs the embedding model (4a) to transform the query into vector embeddings (5a). The next step is choosing the appropriate searching strategy to find the relevant response for the query. The search strategy has evolved from keyword matching to semantic search. Keyword matching looks for specific index words that match the query words, whereas, semantic searching focuses on the meaning of the query words and finds contextually relevant data. The search is performed on pre-existing data, and the number of relevant data points returned is determined by setting the value of *k* in top-*k* similarity search. In our approach, the *VectorIndexRetriever* performs a top-*k* semantic retrieval (where *k*=2) from the *VectorStoreIndex* containing the embeddings of the technical documentation. This retrieval is based on cosine similarity, calculating the likeness between the query's embedding and those of the technical documents. This identifies outdated sections of the documentation by comparing the current technical documentation with recent code changes. The top-*k* retrieved nodes (7a) are then forwarded to the *response synthesis* component (8a). The *response synthesis* component combines the user's query with nodes retrieved in the previous step and sends a prompt to the LLM (9a). Among the various methods for generating LLM responses, our solution uses the *compact response* generation technique. This technique is chosen because it minimizes LLM calls while sending code changes to the LLM by fitting multiple nodes into a single prompt, up to the allowed prompt size. The interaction is performed in a question-answer format using a *create and refine* prompt template from LLamaIndex. The result is the generation of the updated technical documentation (10a).

4.3 MR Summary

The summary feature relies solely on MR data to generate a concise summary of the MR itself. Unlike the previous approach applied for updating the technical documentation where the MR was directly used as a query, the current method involves converting the MR into embeddings and storing them in a *VectorStoreIndex* for future retrieval. The initial step entails indexing the filtered MR data by leveraging the GitLab API, which provides a list of JSON objects representing the modifications within an MR. For each JSON object, a LlamaIndex *document* is created with the main content sourced from the diff text. This *document* is populated with the diff text and enriched with metadata attributes such as ***new_path***, ***old_path***, ***new_file***, ***renamed_file***, and ***deleted_file*** to provide additional context about the code changes.

The number of *documents* corresponds to the number of filtered code change files. Consistently with prior implementations, documents and nodes are generated using a chunk size of 1,024 tokens and a chunk overlap of 20 tokens. If a file's content exceeds 1,024 tokens, it is divided across multiple nodes. These nodes are then processed by the embedding model (B), and the resulting MR embeddings are stored in a *VectorStoreIndex* for later use (E). This embedding process occurs when an MR is selected for analysis. When a technical writer requests a summary of MR changes (1b), the *SummaryIndexRetriever* asks for the relevant nodes (2b) from the *VectorStoreIndex* where they are stored. Unlike the prior method which returned only the top-*k* nodes, here, all nodes are returned (3b) because each contributes to the summary feature. The returned nodes are then passed to the *response synthesis* component (4b). To generate the LLM response, a *tree summarization* generation technique is employed, where nodes are recursively sent to the LLM using a summary prompt template (5b). Depending on the variety of changes, multiple responses may be generated. The final summary is constructed by organizing these responses into a tree structure, with the root node providing a complete summary (6b).

4.4 MR Chat

The MR chat functionality, like the summary feature, relies solely on MR data to perform its operations. When a technical writer poses a question about MR changes through the MR chat (1c), the *VectorIndexRetriever* employs the embedding model (2c) to convert the question into vector embeddings (3c). Following this, a top-*k* semantic retrieval process (4c) occurs within the *VectorStoreIndex*, the repository for MR embeddings generated in advance. Here, only the top-*k* nodes (where *k*=2) are retrieved (1c), as they contain the most relevant segments of the MR data necessary for answering the technical writer's question. These selected nodes are then passed to the *response synthesis* component (6c), which formulates and sends a prompt to the LLM (7c), leveraging a *compact response* generation technique, similar to that used in the query engine. This process ultimately generates an answer to the technical writer's question (8c).

4.5 Large Language Model

The three LLMs selected for this study, as detailed in Table 1, are chosen based on factors such as cost and context window. However, our approach is not tied to these specific models; it is designed to be adaptable, allowing easy integration of additional LLMs via an API endpoint.

Table 1: LLMs and their utilized parameters in our implementation.

Model	Provider	Context Window
gpt-3.5-turbo-0125	OpenAI	16,385 tokens
llama-v2-34b-code-instruct	Meta	16k - 100k tokens
llama-v3-70b-instruct	Meta	8k tokens

- The *gpt-3.5-turbo-0125* model, developed by OpenAI, features a substantial context window of 16k tokens. This model is designed for conversational tasks, allowing it to handle extensive inputs effectively, making it ideal for applications that require a deep understanding of context, such as documentation generation and summarization. The cost is \$0.0005 for every 1,000 input tokens and \$0.0015 for every 1,000 output tokens.
- The *llama-v2-34b-code-instruct* model from Meta

is an open-source model specifically designed for code-related tasks. It features a context window that ranges from 16k to 100k tokens, enabling efficient handling of large merge requests This model is specifically trained for general code synthesis and comprehension, essential for our goal of analyzing diff outputs to identify necessary documentation updates.

- The *llama-v3-70b-instruct model* from Meta is an open-source model that shows excellent performance in reasoning, code generation, and following instructions(Meta, 2024). It features a context window of around 8k tokens, which is sufficient for handling merge requests that are of moderate size.

4.6 Quality Measures

The RAG pipeline forms the backbone of our solution, with separate evaluations performed on both the retriever and response synthesis components using pre-defined metrics.

The *contextual relevancy* metric is used to evaluate the performance of the retriever component. During the retrieval phase, the query is converted into an embedding, followed by a similarity search to locate the most relevant documents from the vector store. This metric is calculated as the ratio of the relevant statements retrieved by the retriever for a given query to the total number of statements.

The *answer relevancy* and *faithfulness* metrics are used to evaluate the performance of the response synthesis component, otherwise also referred to as the generator component. The *answer relevancy* metric measures how relevant the generated answer is compared to the input query. Unlike *contextual relevancy*, this metric is calculated as the ratio of relevant statements generated by the chosen LLM for a given query to the total number of generated statements. The *faithfulness* metric measures the factual consistency of the output by assessing how accurately its content aligns with the information from the retrieved documents. This metric is calculated by first extracting all claims from the generated output. Then, among these claims, those that do not contain any factual contradictions when compared to the retrieved documents are identified as truthful claims. This metric is the ratio of truthful claims to the total number of claims in the generated output.

It is worth noticing that our approach proposes the use of standard metrics measuring the quality of the generated output based on the characteristics of the prompts. In this respect, our approach does not require a ground truth dataset to evaluate the quality of the generated documentation.

5 INDUSTRIAL CASE STUDY

CodeDocSync is applied in Nokia's industrial CSD environment, where frequent code changes require continuous updates to technical documentation. Here, we focus on Nokia's Network as Code (NAC) project to demonstrate CodeDocSync's capability to automate technical documentation updates in response to changes in the Software Development Kit (SDK). Nokia's NAC project enables developers to dynamically control network performance directly from downstream applications. Although Nokia currently employs a docs-as-code approach, this method often requires developers to draft updates and technical writers to finalize them, making the process both time-consuming and inefficient. Given its frequent feature enhancements, the NAC project serves as an ideal case for evaluating CodeDocSync's effectiveness in ensuring accurate, up-to-date documentation. The interested reader can access further details on the implementation code discussed in this paper in the GitHub repository[3].

5.1 Implementation

CodeDocSync is integrated with NAC's SDK and technical documentation repositories, leveraging GitLab's API to automatically retrieve code changes from open MRs and existing technical documentation. The existing documentation comprises approximately 40 technical markdown files, including concise API references, general concepts, and instructions. It features nested structures and domain-specific terminology related to 5G networking, reflecting the complexity nature of the content. The size of the codebase is less than 1000 files. The MR changes are processed through a custom pipeline that filters the original JSON data and performs several pre-processing steps: i) extracting the Git diff, ii) cleaning the data by removing irrelevant changes that introduce noise, and iii) chunking the changes based on file paths. The documentation repository categorizes content into separate markdown files, each dedicated to specific topics, such as *network slicing*. These files are loaded and filtered to exclude unnecessary files, and then chunked into distinct nodes, typically with each node representing a markdown page on a particular topic. After preprocessing, both the

[3]https://anonymous.4open.science/r/codedocsync-85F2/README.md

code changes and documentation nodes are converted to embeddings and stored in a *VectorStoreIndex*, enabling efficient retrieval and alignment during query processing.

For evaluation purposes, we focused on the *network slice attachment* feature within NAC. First, an MR is created in the SDK repository to modify the existing `attach()` and `detach()` methods of this feature. The existing technical documentation, which does not yet reflect these changes, is stored in the *VectorStoreIndex*. As shown in Figure 2, selecting the MR displays a page listing the files that are ready for analysis. This page also includes three tabs - one for each of the core CodeDocSync features.

The *fetch summary* tab provides concise information for each modified file, with a concluding paragraph to give an overall explanation of the changes as shown in Figure 3. The summarization can be customized to focus on specific files by excluding selected files from the first tab.

The *fetch updated documentation* tab provides the updated documentation reflecting the recent changes. As shown in Figure 4, the *new feature?* flag is available to generate documentation for a new feature that has not been previously covered in the existing documentation. If the option *new feature?* is selected, the prompt will be updated to generate new documentation content based on the data from the code changes. The generated content is formatted in Markdown syntax for easier integration into the final documentation.

The *MR-chat* tab is designed to assist the technical writer in understanding changes in the selected merge request by enabling follow-up questions. As shown in Figure 5 , the technical writer can ask questions about details such as the new method signature or request a general explanation of the changes and the chat will provide the required response.

5.2 Evaluation

This section presents the evaluation results for our chosen case study. The evaluation is performed using the DeepEval [4] tool by applying the predefined metrics outlined in Section 4.6, and the OpenAI's *gpt-4o* model.

5.2.1 Contextual Relevancy

The retriever component of the RAG pipeline is evaluated based on contextual relevancy using three different values of k, which represent the number of relevant nodes expected in the top-k similarity search. Figure 6 shows that the retriever component of our solution performs more effectively as the k value decreases. The average contextual relevancy score increases from 0.53 at $k = 5$ to 0.83 at $k = 2$. This improvement suggests that lower k values allow the retriever to focus more precisely on the most relevant nodes, reducing noise from less relevant results.

5.2.2 Answer Relevancy

The answer relevancy metric is used to evaluate the performance of the response synthesis component which in DeepEval is referred to as the generator component with assessments conducted for two independent variables: temperature and LLM - embedding model combination. Temperature is set to ensure creative but accurate outputs. Both contextual and answer relevancy metrics are calculated using Equation 1.

$$Relevancy = \frac{Number\_of\_Relevant\_Statements}{Total\_Number\_of\_Statements} \quad (1)$$

Figure 7 shows the answer relevancy scores for different LLM - embedding model combinations. The combination of the *gpt-3.5-turbo-0125* LLM with the *text-embedding-3-large* embedding model achieves the highest average score of 0.86, which increases to 0.94 when the *emotion-prompting technique* described in Section 4.2.3 is applied to the prompt template[5]. In contrast, the combination of the *llama-v3-70b-instruct* LLM with the *BAAI/bge-small-en-v1.5* embedding model reaches a highest average score of only 0.3. The combination of the *llama-v3-70b-instruct* LLM with the *text-embedding-3-large* embedding model improved the score to 0.68. These results highlight the impact of the LLM - embedding model combinations on performance.

5.2.3 Faithfulness

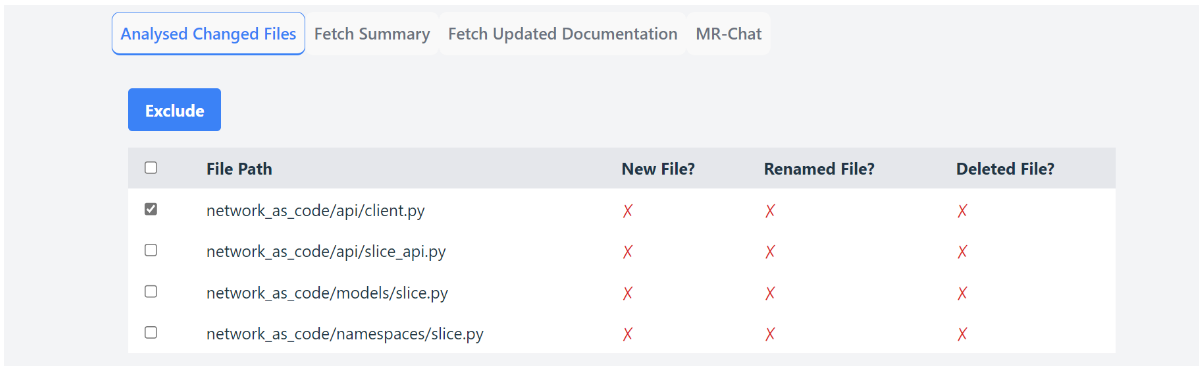
The faithfulness metric is used to assess the factual accuracy of the LLM's response in relation to the content retrieved by the retriever component. The LLM - embedding model combination serves as the independent variable. It is calculated using Equation 2.

$$Faithfulness = \frac{Number\_of\_Truth\_Claims}{Total\_Number\_of\_Claims} \quad (2)$$

Figure 8 shows the faithfulness scores, where the combination of the *gpt-3.5-turbo-0125* LLM with the *text-embedding-3-large* embedding model has an average score of 0.82. This is lower than the

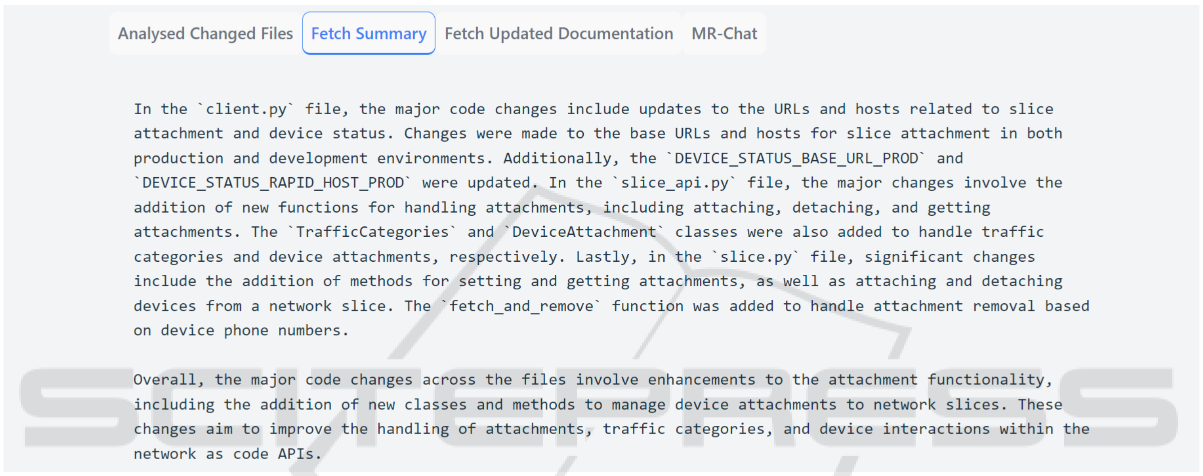
[4]https://docs.confident-ai.com

[5]For the sake of space limitations we do not report the results in full details. The interested reader is referred to (Birru, 2024) for the complete information about the used prompts and corresponding results/performances



☐	File Path	New File?	Renamed File?	Deleted File?
☑	network_as_code/api/client.py	✗	✗	✗
☐	network_as_code/api/slice_api.py	✗	✗	✗
☐	network_as_code/models/slice.py	✗	✗	✗
☐	network_as_code/namespaces/slice.py	✗	✗	✗

Figure 2: List of filtered changes for Network Slice Attachment.



```
In the `client.py` file, the major code changes include updates to the URLs and hosts related to slice
attachment and device status. Changes were made to the base URLs and hosts for slice attachment in both
production and development environments. Additionally, the `DEVICE_STATUS_BASE_URL_PROD` and
`DEVICE_STATUS_RAPID_HOST_PROD` were updated. In the `slice_api.py` file, the major changes involve the
addition of new functions for handling attachments, including attaching, detaching, and getting
attachments. The `TrafficCategories` and `DeviceAttachment` classes were also added to handle traffic
categories and device attachments, respectively. Lastly, in the `slice.py` file, significant changes
include the addition of methods for setting and getting attachments, as well as attaching and detaching
devices from a network slice. The `fetch_and_remove` function was added to handle attachment removal based
on device phone numbers.

Overall, the major code changes across the files involve enhancements to the attachment functionality,
including the addition of new classes and methods to manage device attachments to network slices. These
changes aim to improve the handling of attachments, traffic categories, and device interactions within the
network as code APIs.
```

Figure 3: MR-summary for Network Slice Attachment.

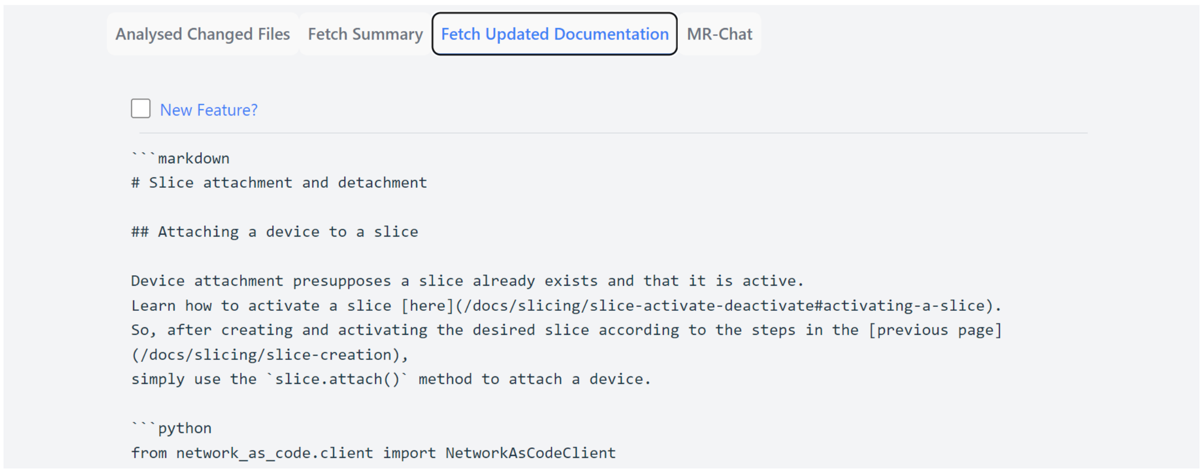
combination of the *llama-v3-70b-instruct* LLM with the *BAAI/bge-small-en-v1.5* embedding model, which achieves an average score of 0.95. This difference may be due to the *llama-v3-70b-instruct* model's better alignment with the retrieved content, leading to more factually accurate responses.

5.3 Discussion

The RAG prompt engineering technique is used in our solution, which enables responses to be generated directly from existing technical documentation and code change data. In addition to RAG, we incorporated advanced techniques such as emotion prompting and expert prompting within the prompt templates used in the RAG pipeline. Initially, we experimented with simpler techniques, such as zero-shot and one-shot prompts; however, given the size of the documentation data and the context window limitations of LLMs, RAG proved to be the optimal approach. This choice not only maximized context utilization but also opened up opportunities to integrate MR-summary and MR-chat features, improving the solution's functionality and user experience.

Our solution updates documentation affected by code changes by embedding the *“diff”* from MRs into the RAG pipeline. To handle the token limits of the embedding model, we filter irrelevant information from MRs and then apply chunking to divide large MR size, preserving all content without exceeding limits. Each chunk's embedding is then combined into a single vector using a weighted average, ensuring proportional representation based on chunk length. This approach optimizes the embedding process, enabling effective similarity searches against technical documentation even when handling large MRs.

Even though a thorough empirical assessment for the value of our proposed automated documentation generation goes beyond the scope of this work, we performed interviews with a technical writer and a developer at Nokia, who tested the tool and provided feedback. While they appreciated the overall precision and usability of the documentation update support, they also suggested the improvement of certain features. Notably, they expressed the need for visual



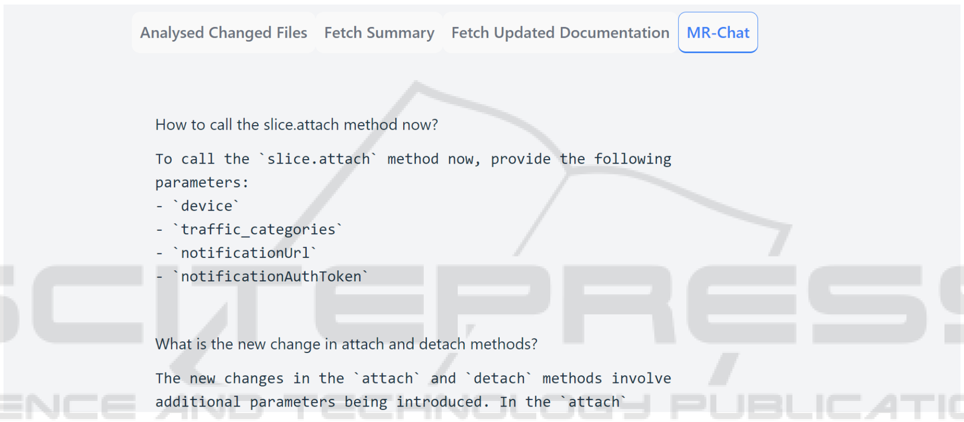
~~~
```markdown
# Slice attachment and detachment

## Attaching a device to a slice

Device attachment presupposes a slice already exists and that it is active.
Learn how to activate a slice [here](/docs/slicing/slice-activate-deactivate#activating-a-slice).
So, after creating and activating the desired slice according to the steps in the [previous page]
(/docs/slicing/slice-creation),
simply use the `slice.attach()` method to attach a device.

```python
from network_as_code.client import NetworkAsCodeClient
~~~

Figure 4: Updated documentation section for Network Slice Attachment.



How to call the slice.attach method now?

```
To call the `slice.attach` method now, provide the following
parameters:
- `device`
- `traffic_categories`
- `notificationUrl`
- `notificationAuthToken`
```

What is the new change in attach and detach methods?

```
The new changes in the `attach` and `detach` methods involve
additional parameters being introduced. In the `attach`
```

Figure 5: MR-chat for Network Slice Attachment.

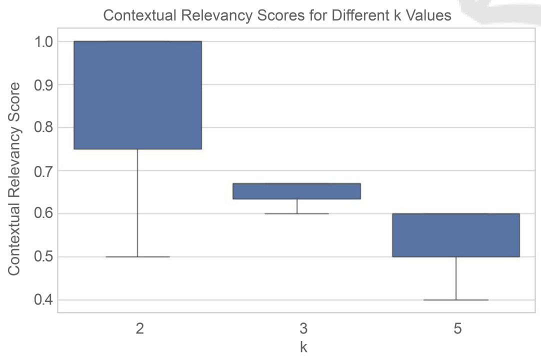


Figure 6: Contextual relevancy score at various *k* values.

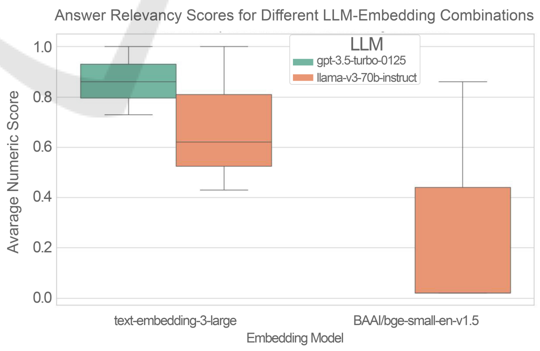


Figure 7: Answer relevancy scores for LLM - embedding model combinations.

indicators or explanations able to distinguish specific changes in updated documentation. Such distinction would significantly improve clarity and usability by allowing to isolate specific changes and corresponding updates.

While our solution primarily focuses on updating existing documentation, some MRs introduce entirely new information not covered in current documentation. To address this, our solution offers a *"New Feature"* flag that, when enabled, allows technical writers to generate documentation for new features. When this flag is active, the system applies a prompt specifically designed to create new content, ensuring documentation coverage for new features.

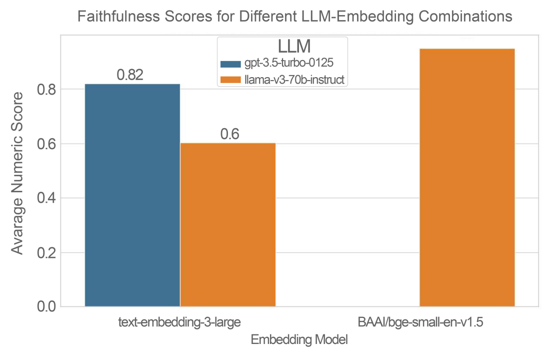


Figure 8: Faithfulness metric scores for LLM - embedding model combinations.

Our approach relies on the quality of source documentation, and its effectiveness may be limited if the documentation is poorly organized, potentially impacting the overall results. In a broader perspective, it would be important to test our solution across diverse project types, especially those with varying programming languages and technical writing styles. Although the solution is not inherently dependent on any specific programming language, the evaluation has been conducted using only Python-based projects. Expanding testing to include projects in other languages would help assess the solution's versatility. In this respect, it is worth noticing that the research community is missing a standard procedure to evaluate the automated generation of documentation. Therefore, establishing standard benchmarking procedures would be valuable to enable the validation and refinement of the proposed solutions.

6 CONCLUSIONS AND FUTURE WORKS

In CSD settings, keeping updated the technical documentation of software is critical, especially when such software is used further in upstream and downstream developments. Despite the availability of techniques to support documentation updates, this task is still largely manual. As a consequence, when software is changed frequently keeping the pace for documentation becomes rapidly intractable.

We presented CodeDocSync, an approach devoted to enhancing documentation updates in CSD. The approach leverages LLMs to provide: an automatic updater of the existing technical documentation based on source code changes; a change descriptor summarizing how the modifications have been interpreted; a chat replying to custom questions posed by technical writers.

The concretization in an industrial case study demonstrates the potentials of the proposed approach in terms of quality of generated documentation and corresponding support. Nonetheless, further investigations are required to empirically measure the gains provided by the automation as well as its applicability to diverse kinds of applications and programming languages. These empirical validations would explore deeper qualitative feedback from domain experts involved in documentation handling.

From a more technical perspective, our approach improves the existing state of the art by avoiding a training phase for the automation support. However, it would be interesting to provide a more structured and detailed comparison between pre-trained models and those fine-tuned on appropriate datasets to measure the gaps in quality of the generation. Additionally, improvements to the RAG system, such as integrating re-ranking mechanisms or advanced indexing techniques, should be explored to enhance performance.

REFERENCES

Aghajani, E., Nagy, C., Linares-Vásquez, M., Moreno, L., Bavota, G., Lanza, M., and Shepherd, D. C. (2020). Software documentation: the practitioners' perspective. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20. ACM.

Aghajani, E., Nagy, C., Vega-Marquez, O. L., Linares-Vasquez, M., Moreno, L., Bavota, G., and Lanza, M. (2019). Software documentation issues unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE.

Ahmad, W., Chakraborty, S., Ray, B., and Chang, K.-W. (2020). A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics.

Ahmad, W. U., Chakraborty, S., Ray, B., and Chang, K.-W. (2021). Unified pre-training for program understanding and generation.

Ahmed, T. and Devanbu, P. (2022). Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. arXiv.

Alephium (2023). AI-Powered Improved Docs: Explaining Alephium Full Node Code — alephium. https://medium.com/@alephium/ai-powered-improved-docs-explaining-alephium-full-node-code-6795667fac02. [Accessed 15-11-2024].

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. (2021). Program synthesis with large language models.

Batarseh, F. A., Mohod, R., Kumar, A., and Bui, J. (2021). The application of artificial intelligence in software engineering: a review challenging conventional wisdom.

Birru, H. (2024). Exploring the use of llms in agile technical documentation writing. Master's thesis, Mälardalen University, Västerås, Sweden.

Bosch, J. (2014). *Continuous Software Engineering: An Introduction*, page 3–13. Springer International Publishing.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. (2021). Evaluating large language models trained on code.

Chui, M., Manyika, J., and Miremadi, M. (2016). Where machines could replace humans-and where they can't (yet). *The McKinsey Quarterly*, pages 1–12.

Dau, A. T. V., Guo, J. L. C., and Bui, N. D. Q. (2023). Docchecker: Bootstrapping code large language model for detecting and resolving code-comment inconsistencies.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages.

Gadesha, V. and Kavlakoglu, E. (2024). What is prompt chaining? — IBM — ibm.com. https://www.ibm.com/topics/prompt-chaining. [Accessed 14-05-2024].

Gao, S., Wen, X.-C., Gao, C., Wang, W., Zhang, H., and Lyu, M. R. (2023). What makes good in-context demonstrations for code intelligence tasks with llms? In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE.

Haiduc, S., Aponte, J., and Marcus, A. (2010a). Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10. ACM.

Haiduc, S., Aponte, J., Moreno, L., and Marcus, A. (2010b). On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*. IEEE.

Holscher, E. (2024). Docs as Code — writethedocs.org. https://www.writethedocs.org/guide/docs-as-code/. [Accessed 28-01-2024].

Hu, X., Li, G., Xia, X., Lo, D., Lu, S., and Jin, Z. (2018). Summarizing source code with transferred api knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, IJCAI-2018. International Joint Conferences on Artificial Intelligence Organization.

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. (2019). Codesearchnet challenge: Evaluating the state of semantic code search.

Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y. J., Madotto, A., and Fung, P. (2023). Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38.

Khan, J. Y. and Uddin, G. (2022). Automatic code documentation generation using gpt-3.

Khan, J. Y. and Uddin, G. (2023). Combining contexts from multiple sources for documentation-specific code example generation.

Li, C., Wang, J., Zhu, K., Zhang, Y., Hou, W., Lian, J., and Xie, X. (2023). Emotionprompt: Leveraging psychology for large language models enhancement via emotional stimulus. *ArXiv*, abs/2307.11760.

Liu, Z., Xia, X., Lo, D., Yan, M., and Li, S. (2023). Just-in-time obsolete comment detection and update. *IEEE Transactions on Software Engineering*, 49(1):1–23.

Lu, J., Yu, L., Li, X., Yang, L., and Zuo, C. (2023). Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 647–658.

Luo, Q., Ye, Y., Liang, S., Zhang, Z., Qin, Y., Lu, Y., Wu, Y., Cong, X., Lin, Y., Zhang, Y., Che, X., Liu, Z., and Sun, M. (2024). Repoagent: An llm-powered open-source framework for repository-level code documentation generation.

Mastropaolo, A., Cooper, N., Palacio, D. N., Scalabrino, S., Poshyvanyk, D., Oliveto, R., and Bavota, G. (2023). Using transfer learning for code-related tasks. *IEEE Transactions on Software Engineering*, 49(4):1580–1598.

Meta (2024). Introducing Meta Llama 3: The most capable openly available LLM to date — ai.meta.com. https://ai.meta.com/blog/meta-llama-3/. [Accessed 03-06-2024].

Moore, J., Gelman, B., and Slater, D. (2019). A convolutional neural network for language-agnostic source code summarization. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering*, ENASE 2019, page 15–26, Setubal, PRT. SCITEPRESS - Science and Technology Publications, Lda.

Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L., and Vijay-Shanker, K. (2013). Automatic generation of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE.

Nguyen-Duc, A., Cabrero-Daniel, B., Przybylek, A., Arora, C., Khanna, D., Herda, T., Rafiq, U., Melegati, J., Guerra, E., Kemell, K.-K., Saari, M., Zhang, Z., Le, H., Quan, T., and Abrahamsson, P. (2023). Generative artificial intelligence for software engineering – a research agenda.

Phan, L., Tran, H., Le, D., Nguyen, H., Anibal, J., Peltekian, A., and Ye, Y. (2021). Cotext: Multi-task learning with code-text transformer.

Rai, S., Belwal, R. C., and Gupta, A. (2022). A review on source code documentation. *ACM Transactions on Intelligent Systems and Technology*, 13(5):1–44.

Sahoo, P., Singh, A. K., Saha, S., Jain, V., Mondal, S., and Chadha, A. (2024). A systematic survey of prompt engineering in large language models: Techniques and applications.

Shin, J., Tang, C., Mohati, T., Nayebi, M., Wang, S., and Hemmati, H. (2023). Prompt engineering or fine tuning: An empirical assessment of large language models in automated software engineering tasks.

Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., and Vijay-Shanker, K. (2010). Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE10. ACM.

Su, Y., Wan, C., Sethi, U., Lu, S., Musuvathi, M., and Nath, S. (2023). 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*, HOTOS '23. ACM.

Tan, W. S., Wagner, M., and Treude, C. (2023). Wait, wasn't that code here before? detecting outdated software documentation. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE.

Tan, W. S., Wagner, M., and Treude, C. (2024). Detecting outdated code element references in software repository documentation. *Empir. Softw. Eng.*, 29(1).

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., and Zhou, D. (2022). Self-consistency improves chain of thought reasoning in language models.

Wen, F., Nagy, C., Bavota, G., and Lanza, M. (2019). A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE.

Xu, B., Yang, A., Lin, J., Wang, Q., Zhou, C., Zhang, Y., and Mao, Z. (2023a). Expertprompting: Instructing large language models to be distinguished experts.

Xu, B., Yang, A., Lin, J., Wang, Q., Zhou, C., Zhang, Y., and Mao, Z. (2023b). Expertprompting: Instructing large language models to be distinguished experts.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., and Narasimhan, K. (2023). Tree of thoughts: Deliberate problem solving with large language models.

Zheng, Z., Ning, K., Chen, J., Wang, Y., Chen, W., Guo, L., and Wang, W. (2023). Towards an understanding of large language models in software engineering tasks.