Design of Cross-Repository API Documentation Automation for Projects Governed by Version Control

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Software Engineering
June 2025
Valtteri Sjöblom

Supervisors:
Ville Leppänen

UNIVERSITY OF TURKU
Department of Computing

VALTTERI SJÖBLOM: Design of Cross-Repository API Documentation Automation for Projects Governed by Version Control

Master of Science (Tech) Thesis, 51 p., 3 app. p.
Software Engineering
June 2025

Proper API documentation is vitally important for developing with the API. It should be provided in a clear and understandable format, with heightened accessibility and accuracy. This thesis is an attempt to ease the production of API documentation, by automating the process of its creation in a multi-repository project structure. The thesis will also survey the current state of generative AI usage within this space, and acquire knowledge on how following improvements for the implementation of this design can utilize AI. This, and finding a way to organize this automation system, such that it could scale for the inclusion of other projects, are the research objectives of this thesis. Creating the design and its implementation follow Design Science Research, while the literature review follows an exclusion method to find relevant articles for research within context. The design can be presumed to scale well with the anticipation of new projects, although a definitive and objective answer was not possible to be deduced from the limited data gathered. There were a few possibilities on how to include AI within this design, but none are mature enough yet to be used without human intervention. More research on the AI side is still necessary. The automation system is in production, and more data will be gathered from it in-house.

Keywords: software, documentation, automation, AI, LLM, API, generative, version control

Contents

List of Figures

List of Tables

List of code snippets

List of acronyms

AI Artificial Intelligence

API Application Programming Interface

AST Abstract Syntax Tree

CI/CD Continuous Integration/Continuous Delivery

DAG Directed Acyclic Graph

HTML Hypertext Markup Language

IDE Integrated Development Environment

IT Information Technology

JSON JavaScript Object Notation

LLM Large-Language-Model

NDA Non-Disclosure Agreement

RAG Retrieval-Augmented Generation

REST Representational State Transfer

SSH Secure Shell

UML Unified Modeling Language

1 Introduction

During a rewrite project of a web portal, a lack of proper API documentation became an apparent problem, which caused unnecessary duplication of reusable utilities and components, confusion on their function, and slowness of development, mainly due to the constant probing of the code base. A fix for this could be rather easy; disciplined documentation practices. However, this web portal is decently sized, with tens of thousands of lines of code, and while discipline is necessary in its own right, creating a simplified process to build and serve a usable API documentation eases the burden of developers, increasing the chance that all the necessary API documentation gets written. This thesis is an attempt to fill this void by designing (and implementing) a quasi-novel API documentation automation system built upon existing tooling, which would support multiple projects and their branches, while simultaneously providing the documentation for users of previous versions of an API.

1.1 About Research

The research for this thesis was structured around two distinct methodological approaches, where each of them addresses a specific research question. To enable continuity and future research and development possibilities on the design of this documentation system, a lightweight literature review on the use of generative AI for alteration, creation and utilization enhancement of API documentation was conducted. The design itself is not utilizing these AI powered features yet, but considerations on how they might be implemented is discussed. The other research approach utilizes Design Science Research to implement the API documentation automation system as a part of the web portal rewrite project. The research questions these methods will answer are as follows:

RQ1 How to effectively organize API documentation generation and delivery, so that it:

a. scales with multi-repository project structures?

b. supports multiple versions simultaneously?

RQ2 How can generative AI help developers with API documentation maintainability and usability?

The objective of Design Science Research is to develop knowledge about problems professionals face in real world applications, by developing an artifact for evaluation. The process of research in Design Science Research defines 6 activities for researchers; problem identification and motivation, defining

objectives for a solution, design and development of a solution, demonstrating the result, evaluating its performance, and communicating the results. The process is iterative, such that after the evaluation phase, the researchers can decide to go back to redesigning the solution, or move on to communicating the results and leave further improvements to subsequent projects. [1]

1.2 Thesis Relevance

This study was conducted within the context of a company operating in the field of cyber-security and networking. While the design of an API documentation system is in itself not highly confidential material, the work is done under NDA, which prevents discussions of technical details about the underlying application for which the design of this thesis is used in, and the design itself. This design uses readily available open source tooling, and could therefore be replicated in any similar environment with decent professional knowledge and expertise. If it performs and scales adequately, it could be used as the backbone of all API documentation within the company.

There are countless studies from multiple angles on API documentation; automation and systems [2] [3] [4] [5] [6], goodness evaluation [7] [8], best practices [9] [10] [11], and so on. In contrast, as the use of LLMs as generative AI boomed after OpenAI released its GPT-3 model and the ChatGPT service circa 2022 [12], the area of study around their usage in a wide range of applications, such as API documentation generation, is rather new. In this thesis, the literature review examines the current state of research within the

specific subset of AI usage. While not its primary goal, this review uncovers gaps for future research. The primary goal for the review is to answer RQ2, and guide the following design of using generative AI for API documentation enhancements in the future.

1.3 Thesis Overview

After this introductory chapter, Chapter 2 equips the reader with foundational concepts required to navigate the rest of this thesis. It is assumed that the reader possesses prior knowledge of fundamental IT concepts not directly relevant to the subject matter. This fundamental knowledge is thus not discussed in detail.

Chapter 3 explains the methodology for the literature review on the use generative AI within the scope of the thesis, and presents the findings. What follows, is a brief reflection on how AI could be utilized within the documentation space, answering RQ2.

Chapter 4 dives in to the design of the API documentation generation system. Using the Design Science Research methodology, it starts by explaining the context, its constraints and requirements, and the implications these have for the design itself. The following sections provide as detailed depiction of the design as possible under the limitations of the NDA, and continue to demonstrate how it works in practice.

In Chapter 5 the design is assessed based on criteria defined and constrained to help answer RQ1a and RQ1b. Chapter 6 concludes this thesis by reflecting on the knowledge and insights gained through the research, and

explores the future possibilities for this design.

2 Background

This chapter discusses the relevant background knowledge of API documentation, specifically relating to the automation problem at hand, while also touching on the issues of AI within the context of software engineering. The chapter should equip the reader with adequate knowledge to understand the concepts discussed later on in this thesis. To clarify, API stands for Application Programming Interface, and is used for sharing information between software in all contexts, not only RESTful web services and web applications, but also utility libraries and between two separate programs. ISO/IEC/IEE Standard 26514:2022 [13] states that an API is a *"set of functions (3.1.23), protocols, parameters, and objects of different formats, used to create software (3.1.46) that interfaces with the features or data of an external system or service"*.

2.1 Characteristics of API Documentation

Creating a high quality and usable API documentation is not an easy task to achieve. It requires meticulous attention to detail, and stable practices in its maintenance. Myers and Stylos discuss in their paper [14] of the difficulties of

using APIs; how even the experienced developers have struggles in handling them, leading to bugs in code due to incorrect usage, while still spending substantial amount of time learning the API. This affects the productivity of the developer using the API, leading to many types of problems ranging from psychological to material. The frustration of using an unfamiliar API for development has often lead my co-workers and I to criticize the documentation for it, either due to missing or outdated information, uninformative or incorrect examples, and misleading comments. Noted by Meng et al. [15], API documentation is the main obstacle for learning a new API, a statement which is based on the work of Robillard & DeLine [16] and others. They found five distinct obstacles related to the complexity of an API:

- Background of the developer is not adequate enough to understand the context of the API.
- Structure or design of the API is convoluted.
- Technical environment is not properly set up for using the API.
- The level of documentation is low, resulting in difficulties understanding the context of its usage.
- The level of documentation is high, resulting in difficulties understanding how to apply the code in practice.

These obstacles can all be made easier to overcome by improving the API's documentation; introducing a beginner to the basic concepts related to the API in an introductory chapter, creating detailed descriptions and working examples for the API within the documentation, documenting a

basic setup required to test the API, and generally having different levels of detail within the documentation by for example collapsing the more detailed parts of it. When designing API documentation, the audience of it must be taken into consideration. This is in part to an observation made from the list above, that documentation meant for a beginner might look entirely different from a power-user's documentation. In an open source project, the requirements for allowing beginners to work with the API can vastly differ from projects within a company, where there are co-workers guiding through the usage of the API.

Meng et al. [15] state, that there are two distinct mindsets when approaching API documentation: orientation and learning. Here, orientation means finding out whether the API offers solutions to resolve a problem, and learning means solving a specific problem with the API. I would add to this list from my own experience a third mindset: lookup, which refers to the process of locating the exact or an alternative signature of a specific and familiar function, class or interface. Most modern IDEs do provide quick reference for this type of checking as inlay hints, but referring to the actual documentation has proven helpful often times.

For creating a usable and effective API documentation for each levels of users, Meng et al. [15] provide a 12-point list of detailed guidelines for effective API documentation, based on three heuristics, of which especially the third heuristic *"Support different strategies for learning and development"* supports the observation of the necessity to consider the target audience of the documentation. When other methods of guidance are present, such as the case with co-workers guiding through the initial learning phase, documen-

Heuristic 1	Enable efficient access to relevant content.
Guideline 1.1	Organize the content according to main usage scenarios supported by the API and typical tasks that developers will solve with the API.
Guideline 1.2	Present important conceptual information integrated with the description of tasks or usage scenarios where knowledge of these concepts is needed.
Guideline 1.3	Provide transparent and consistent navigation options and a powerful search function. If integrating search is difficult, e.g. due to technical limitations, facilitate simple search by presenting information on a single page instead of distributing the information across multiple linked pages.
Guideline 1.4	Structure the content at section level consistently. Use appropriate verbal and visual signaling techniques to make the structure transparent.
Heuristic 2	Facilitate initial entry into the API.
Guideline 2.1	Provide clean and working code examples. Code examples need to be complete and accurate and should enable direct re-use via copy-and-paste.
Guideline 2.2	Provide relevant background knowledge on the domain covered by the API and explain important concepts that can support the learning process.
Guideline 2.3	Support developers in relating concepts to API elements. Signal to the developers how concepts are represented, e.g. by emphasizing relevant elements in a code example.
Guideline 2.4	Provide a concise API overview that tells developers and other stakeholders the purpose of the API, its main features and important technical characteristics. Make sure the overview can be accessed easily.
Heuristic 3	Support different strategies for learning and development.
Guideline 3.1	Enable selective access to code, e.g. by using a multi-column layout and formatting means that clearly separate text from code.
Guideline 3.2	Signal text-to-code relations in order to help developers mapping concepts to code. This reduces reading efforts and facilitates the identification of relevant information.
Guideline 3.3	Provide important conceptual information redundantly. Present it wherever needed. If possible, present important conceptual information as part of source code comments to make sure developers focusing on code will discover and process it as well.
Guideline 3.4	Enable fast and productive use of the API. Include code examples and integrate try-out functionality that can be used to test API elements immediately without much effort

Table 2.1: API Documentation Guidelines [15]

tation of the beginner-friendly introduction or the description of an initial technical setup may be omitted. The guidelines presented in Table 2.1 are used implicitly during the construction of the design of this thesis, but they are only indirectly related to the automation problem at present. Heuristic 1, and the guidelines to achieve it are be used mainly during the content creation part by setting up the generator so that the structure it creates is logical and has navigation. The guidelines for heuristic 2 are most important during the writing the documentation. The site generator has to consider all of these heuristics, but especially the third, as it implies the requirements for functionality such as collapsible sections, layouts, links and so on.

Relating to the guidelines, the underlying problems of API documentation they try to mitigate are presented in Figure 2.1; a direct image capture from the article by Uddin and Robillard [7], where they discuss the aspects of API documentation which lead to unusable APIs. While most of the problems specified in the table are the consequence of a lack of discipline in documentation maintenance, following the heuristics can help in alleviating the issues. As an example, heuristic 3 and its guidelines can be used to reduce the documentation bloat by hiding essential excessive knowledge a beginner might need.

Uddin and Robillard [7] found that incompletess, incorrectness and ambiguity are the most important issues that need attention in documentation maintenance. A large part of documentation maintenance is identifying which parts of it need updating. Out of the ten depicted in the figure, they found that the three abovementioned characteristics are the ones that require technical expertise the most. A deep understanding of the underlying

API is required to write accurate documentation for it, thus necessitating the allocation of developer resources for it, ergo time and attention.

Penetrability, another important characteristic of API documentation discussed in [16], is an indicator of API usability in a spectrum of information hiding. Exposing some internals of an API through documentation does at least make the users of it feel as though they understand the API better. However, an excessive need for understanding the internals of how an API

Category	Problem	Description
Content	Incompleteness	The description of an API element or topic wasn't where it was expected to be.
	Ambiguity	The description of an API element was mostly complete but unclear.
	Unexplained examples	A code example was insufficiently explained.
	Obsoleteness	The documentation on a topic referred to a previous version of the API.
	Inconsistency	The documentation of elements meant to be combined didn't agree.
	Incorrectness	Some information was incorrect.
		Total
Presentation	Bloat	The description of an API element or topic was verbose or excessively extensive.
	Fragmentation	The information related to an element or topic was fragmented or scattered over too many pages or sections.
	Excess structural information	The description of an element contained redundant information about the element's syntax or structure, which could be easily obtained through modern IDEs.
	Tangled information	The description of an API element or topic was tangled with information the respondent didn't need.

Figure 2.1: Image capture of Uddin and Robillard Table 2: API documentation problems reported in the exploratory survey [7]

works may be a design smell for the API itself, as they are generally used to abstract away implementation details. Robillard and DeLine [16] found that developers tend to extract information from the source code when its documentation was lacking. Due to finding myself doing this occasionally, I discussed this practice with colleagues, and we came to the conclusion that such a practice may lead to the misuse of the API from exploiting an undocumented specific scenario the code allows. Relying on such cases may lead to problems in the future, when the API is updated. This highlights the importance of documentation accuracy, and the need for explaining relevant side-effects of functionality to limit the reliance to source code.

2.2 About Content Creation

There are a number of ways to write API documentation; what to include, which formats to use, how to organize the presentation, and so on. From a technical standpoint while considering automation, the primary way to create documentation has been to include it within the source code. Per Wikipedia [17], this practice has been developing since at least 1989, with the first publicly released documentation generator used to extract documentation from comments called mkd (mkdoc orig.). When writing API documentation inline with the code, these generators may infer information from the code itself, including it in the final documentation. This in turn reduces the amount of documentation that has to be written, as for example type information can be omitted in certain cases. Therefore, modern tooling are generally language specific.

Snippet 1 A Snippet from Golang's *std/bufio* Package [18] as an Example of Code Summary

go

```
// NewReaderSize returns a new [Reader] whose buffer has at least the
// specified size. If the argument io.Reader is already a [Reader] with
// large enough size, it returns the underlying [Reader].
func NewReaderSize(rd io.Reader, size int) *Reader {
    // Is it already a Reader?
    b, ok := rd.(*Reader)
    if ok && len(b.buf) >= size {
     return b
    }
    r := new(Reader)
    r.reset(make([]byte, max(size, minReadBufferSize)), rd)
    return r
}
```

As an example, a code summary is a description of a block of code, that explains how the block can be used, how it should not be used and whether there are some edge-cases that the user should be aware of. This block of code can be either a function, a class, an interface, etc., which gets exposed as an API. The code summary should result in a piece of documentation within the API documentation, replacing the need for reading the actual code. Snippet 1, taken from the Golang standard library, demonstrates this in practice. The block of text above the function definition is considered as the summary. It explains the necessary details of the function in plain

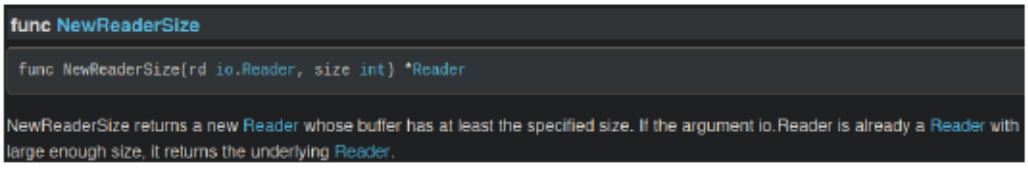

Figure 2.2: Image capture of Snippet 1 Result in Golang Reference Documentation [19]

Snippet 2 Example Story from Storybook Documentation [20]

typescript

```
import type { Meta, StoryObj } from '@storybook/your-framework';

import { Button } from './Button';

const meta = {
  component: Button,
} satisfies Meta<typeof Button>;

export default meta;
type Story = StoryObj<typeof meta>;

export const Primary: Story = {
  args: {
    primary: true,
    label: 'Button',
  },
};
```

text, which gets written to the final documentation page as shown in Figure 2.2. Notice, how the types are inferred from the code and presented in the documentation. These types result in links to the pages describing them.

Another style of API documentation that has been rising in popularity with the advances of today's technology is interactive API documentation, where the API can be tested and validated from the documentation itself. What this means for example for RESTful APIs, is that their usage can be embedded into the documentation, such that a call to the API can be performed from within the documentation, enabling the observation of the API in practice. For UI libraries, this means that a component and its properties can be altered, enabling the observation of how each property affects the out-

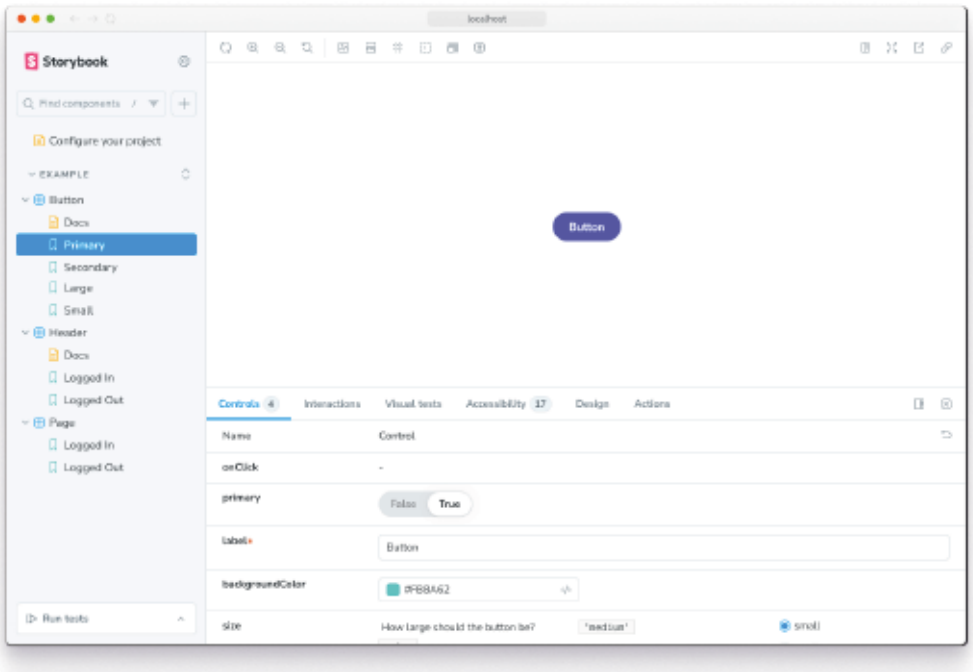

Figure 2.3: Result of the Example Story in Snippet 2.2 [20]

look of the component, or how embedding a component into another works. Storybook [21] is an example of such a system. This style of documentation, especially Storybook, might require more effort and expertise in technical writing and added maintenance compared to the traditional commenting of code as the documentation itself becomes code as seen in Snippet 2.2. Figure 2.3 demonstrates the result of this example snippet, where different options can be manipulated interactively.

2.3 Git & Ecosystem

As git [22] is the de facto version control system used for software projects, an introduction in a technical software related thesis such as this feels redundant.

A 2023 survey from StackOverflow [23] found, that a staggering 93.87% of developers use git, indicating the immense popularity of it compared to other similar systems. Nevertheless, it is an open source distributed version control system with a rich ecosystem built around it. Git works roughly by creating snapshots of the filesystem under version control, and saving references of the files within it. When files have not changed, the references get linked from previous commits. A commit is a term used with git, which can be considered as this snapshot a developer can manually create.

The distributed nature of git allows a developer to work completely locally, as most actions done with git are local. This means, that a connection to a remote upstream repository is not required while working on a project. Only when syncing changes with the remote a connection is needed.

A remote repository behaves exactly like a local repository in its functionality, only that it is generally accessible through a network connection. There are currently multiple service providers for a centralized remote repository management, from which the biggest are probably GitHub, GitLab and Bitbucket. A quick glance at a Google search does not reveal any surveys on service popularity, but multiple reviews and comparisons between the above-

Figure 2.4: Popularity of Git From a StackOverflow Survey [23]

mentioned and others can be found. Open source self-hosted services are also available, each with their own positive and negative aspects.

An important part of a hosting service for git is the availability and implementation of CI/CD pipeline support. CI/CD, which stands for Continuous Integration/Continuous Delivery, allows developers to set up automation for different mundane tasks that have to be performed often or are dependent on some special setup or environment variables, such as tests, builds and deployment. With the pipelines, these tasks can be offloaded from the local development environment, freeing up resources from the local machine to be used for development. As an example implementation, GitHub's version of the CI/CD pipeline system, dubbed as GitHub Actions [24], allows for multi-staged, matrix like workflows, where dependencies can be explicitly stated. With these multi-staged workflows, an important part of the system is the ability to use artifacts created on one stage in another stage to reduce the amount of duplicated work the pipeline has to perform.

These hosting services usually come with a plethora of different ways to ease the development cycle of a project, where a majority of the functionality is non-essential. The main usage for git and the ecosystem is to have version control and automation of mundane tasks via the CI/CD pipelines. These are an integral part of modern software development. The documentation automation system under design of this thesis relies heavily on the functionalities of the git ecosystem.

2.4 Static Site Generator

A static site generator in the context of this thesis, is a tool that generates a full web site ready to be served from certain data or templates [25]. The result of this generation is generally a set of static HTML files, flared with JavaScript and CSS depending on the choice of the generator and the content one wishes to provide. As the pages are mainly static HTML pages, they are quick to serve from a web server host and loaded fast in users' browsers. This is an advantage for an API documentation site with thousands of pages, as rendering each page dynamically would incur drastic overhead and following performance loss in the system compared to statically loading a pre-rendered page. A blog by Phil Hawksworth [26] highlights the improvements in static site generation systems recently, and provides great detail on how to select a generator for different purposes.

From an automation standpoint, these generators are effective tools for easily creating new and updating old content, especially in the case where the content itself is mostly static, as in there are no dynamic elements which would change based on some runtime variable. Hawksworth [26] claims, that since the content is static, it greatly reduces the complexity of serving it. This has an added security benefit, as simpler systems are inherently easier to make secure. There are no databases to attack and the system has reduced amount of logical operations in the chain due to having only a single point where the content is served.

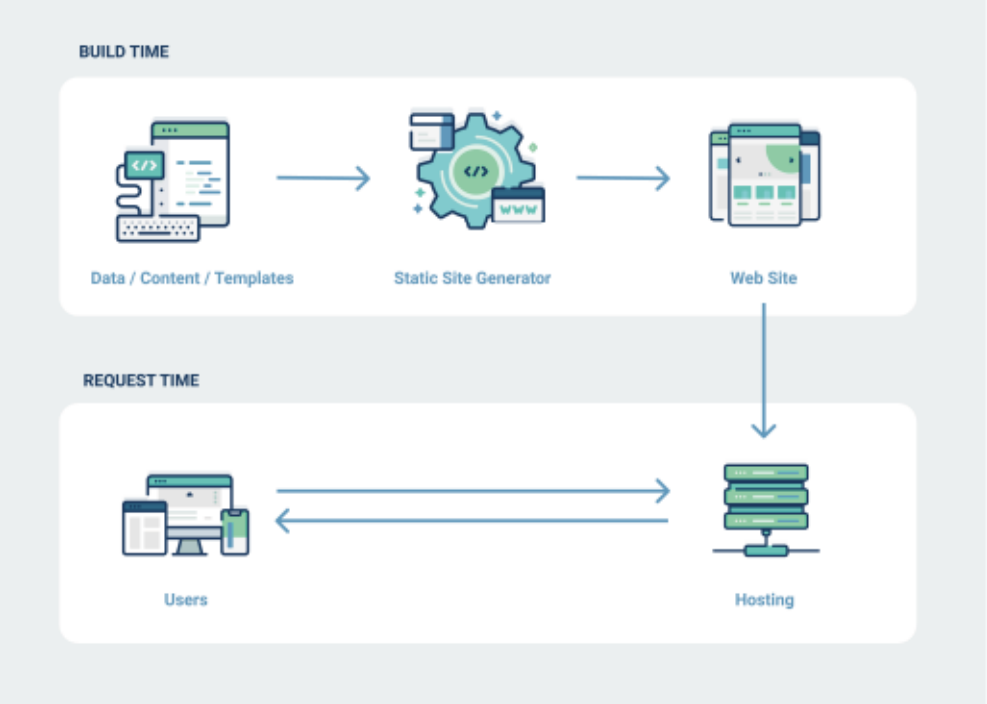

Figure 2.5: Working Principle of Static Site Generators [26]

2.5 Issues of Generative AI in Software

Liutkevičius in his 2025 blog post [27] lists the biggest and most common challenges LLMs face currently within a broader spectrum of usages. From a software development standpoint, inaccuracies and hallucinations are the main concern for its usage, leading to mistrust of the generated output, while simultaneously ethical issues such as copyrights and privacy may be violated when using the generated output. The models are trained with pre-existing material, usually open source projects where a lot of it are subject to licensing which disallows direct and unrestricted reuse. When the model uses the material, it may inadvertently use some of the copyrighted content, creating a possibility for lawsuits.

There are variations upon variations of licenses used for open source, but they are generally categorized by either permissive or copyleft licensing categories. An article by Liran Tal [28] explains how these categories differ, by stating that copyleft licenses will pass down the license to code derived from it, essentially forcing the code to inherit the license, while permissive licenses permit more or less freedom to reuse, modify and distribute the code. As a singular example out of many without providing any factual assertions regarding legal matters, the copyleft GNU General Public License (GPL) [29] states that *"The work must carry prominent notices stating that you modified it, and giving a relevant date"*, meaning that unless use with modifications is not explicitly disclosed, a copyright infringement may result. It is not straightforward to determine whether a piece of AI generated code belongs under such a license.

The pervasiveness of hallucinations is recognized within the scientific and professional communities, as noted by Fan et al. in their survey [30] of open problems in LLM usage for software engineering. Hallucinations specifically mean, that the generative AI creates fictitious output, usually companied with affirmations of its correctness. Fan et al. found in their survey that in an attempt to sidestep the hallucination problem using a code completion suggestion tool powered by an LLM to generate the suggestions, 22% of the suggestions were accepted. Whether this is considered good or not is subject to one's biases, but clearly shows that not all of the suggestions are usable. This relatively low acceptance ratio could be either due to hallucinations, or mere incorrect or inaccurate suggestions. This was not immediately evident from the quantitative study it refers to, done by Murali et al. [31].

3 Survey of Generative AI Use

To provide a foundation for future development on the use of AI within the documentation automation system, a lightweight survey of literature on the use of generative AI within API documentation context was conducted. This chapter first describes the method used for selecting the articles for this review, and then continues to present the findings of the review based on the relevancy for the context of this thesis. The chapter is concluded by a brief summary of the results and discussion on potential use cases for the system.

3.1 Methodology

The process of finding relevant articles discussing the use of generative AI or LLMs was done by an iterative exclusion method. An initial pool of articles was formed by searching academic databases with a predefined keyword, from which the final set of articles were selected by excluding the most irrelevant ones iteratively. The search criteria was: *(llm OR generative or ai) AND documentation*. The results were sorted by relevancy due to the vast amount of results with the query, due to the boom in AI research after the introduction of GPT-3. Prior to its release, the article's likelihood of con-

sidering the most advanced form of generative AI currently available (as in the Large Language Models) starts to decline rapidly [30]. This fact lead to limiting the search to include only articles published after 2022.

Scientific article databases used in this survey were IEEE Electronic Library [32], Science Direct [33], Scopus citation database [34] and ACM Digital Library [35]. To keep the survey lightweight with an aim of reducing the amount of articles used in this literature review to roughly 10, the query result set was further reduced by truncating it to include only the first hundred articles. From these articles, the ones with no relevancy for documentation in their title or abstract were instantly excluded. The resulting set of articles were further narrowed down by inspecting their abstracts, first by providing the set for GPT-o3 in deep research mode to exclude the articles which were not related to AI use in software documentation, to then verify the exclusion through professional assessment of the article's abstract. From this inclusion set, the GPT-o3 was prompted to further exclude similar works based on the overlaps in experiments, datasets or contributions. Again, a professional assessment was used to verify the result. Final inclusion set was selected by assessing whether the articles had novel ideas for AI usage within the context of this thesis by their full content.

Database	**Results**	**Initial**	**In Review**
IEEE Electronic Library	399	17	5
ACM Digital Library	38 723	4	0
Science Direct	12 061	2	0
Scopus	1324	8	2
	Duplicates:	3	
	Total:	28	7

Table 3.1: Results of the Initial Article Exclusion Method

	Articles	Used
GPT Included	14	6
GPT Excluded	14	1
GPT Introduced	7	4

Table 3.2: Results of the GPT-o3 Exclusion Process

Table 3.1 presents the resulting set of articles in numbers, where the "Results" column is the total number results from the search query per database, the "Initial" column indicates how many relevant articles there were in the set of a 100 based on the criteria described above, and the "In Review" column shows the amount of articles selected from each database that were used in the final review. This process was done in May of 2025. The query results included many irrelevant articles from wide range of professions, especially from the field of healthcare research. Three of the articles had duplicates, which were removed. By observing the quantities of initial results versus the sizes of the pools, a slight negative correlation can be inferred, signifying the impact of noise in the results. However, as the pool was over the target set for the number of articles, the initial search was considered a success, and the exclusion procedure continued.

Final set of articles amounted to 11. The results of the first stage of exclusion using GPT-o3 are shown in Table 3.2. The second phase of the GPT-o3 exclusion did not work as intended, as the model disregarded the previous set and used information outside of it. However, it found several interesting new articles for this context, which were included into consideration. Table 3.3 shows the final set of articles included in this review.

Ref	Title	Year
[30]	Large Language Models for Software Engineering: Survey and Open Problems	2023
[36]	Co-AI Technical Writing: Documentation, Experimentation, User Testing, & Ethical Design	2023
[37]	Wait, wasn't that code here before? Detecting Outdated Software Documentation	2023
[38]	Enhancing Software Engineering with AI: Key Insights from ChatGPT	2024
[39]	Automated API Docs Generator using Generative AI	2024
[40]	On mitigating code LLM hallucinations with API documentation	2025
[41]	When LLMs Meet API Documentation: Can Retrieval Augmentation Aid Code Generation Just as It Helps Developers?	2025
[42]	Gorilla: Large Language Model Connected with Massive APIs	2024
[43]	Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning	2023
[44]	RepoAgent: An LLM-Powered Open-Source Framework for Repository-level Code Documentation Generation	2024
[45]	Testing the Effect of Code Documentation on Large Language Model Code Understanding	2024

Table 3.3: Included Articles for the Literature Review

3.2 Survey

While not a novel technique or have a deep analysis on LLM use within the documentation space, Fan et al. discuss the state of LLM usage in software engineering in their 2023 survey [30], which provided baseline knowledge for this survey. Fan et al. state that the focus on LLM-based software engineering has been on generating code rather than documentation, but acknowledge the potential for its use. They identified four promising studies with a focus on code summarization, a prequel to full documentation, and speculated that LLMs may have advantages in this field over other automated methods. Also noted is that our ability to compare and evaluate richer summaries generated

by LLMs may be hindered.

Duin et al. in their panel report [36] discussed the use of AI as a co-author in technical documentation writing, stating that the roles of technical writers will change with Generative AI to become that of an "ideas" person, allowing the AI to do the writing. They also acknowledge that the hype around generative AI technologies may be separated from reality, and advocate for proper communication of its limitations. The paper is rather void of content without going deeper into the references, but does highlight the current interest in the documentation side of generative AI use and the existence of problems of the growing "GenAI Hype", which does affect this space as well.

Tan et al. propose a technique of detecting outdated code element references in project documentation using Github Actions and pull requests in their 2023 article [37]. By examining the top 1000 most popular projects in Github, they found that 28.9% of projects have outdated documentation in some degree. By using their *DOCER_TOOL*, they were able to detect these discrepancies to some degree, realizing the possibility of false positives. While not utilizing generative AI in their work, this study reveals a potential avenue for it. Their findings in the related works did not reveal any other similar tool using generative AI either.

A case study by Al-Ahmad et al. [38] explored how ChatGPT can assists in software engineering tasks, including documentation development. Their method was to provide project-specific details, such as URL paths, parameter descriptions, and edge cases for ChatGPT to provide an outline for documentation. According to the authors' findings, ChatGPT provides a clear and consistent structure for documentation, while simultaneously re-

ducing manual effort of its creation. Despite the upsides, they speculate that the generated documentation may lack detail and nuance on a level required for documenting complex systems, stating that human judgement is still required for documentation generation for accuracy.

Dhyani et al. dive deep into the AI assisted API documentation generation with their work on fine-tuning a model based on *TinyPixel/Llama-2-7B-bf16-sharded* model [39]. They found their optimized model outperforming the baseline model in several categories. Speed of creating an instance of API documentation went from an average of 50 seconds to the average of 36, and the quality of the documentation was improved, being more precise, intelligible and appropriate for the specified scenario. This study, while not explicit on the quality standards, highlights the need for improvements in the models themselves.

A benchmarking study on LLM API hallucinations by Jain et al. [40] revealed, that Retrieval-Augmented Generation (RAG) , dubbed as Documentation Augmented Generation (DAG), improves low-frequency API accuracy, while significantly decreasing high-frequency API accuracy on correct API call generation from cloud-based APIs, using their CloudAPIBench benchmarking tool designed to evaluate API hallucinations. Here, the frequency of an API means how often the API changes or gets used. This DAG process lets an LLM use API documentation before generating a response. They discussed improving the DAG method by selectively triggering the process based on prior knowledge. These trigger rules included an index lookup to check whether the invoked name exists in the API documentation, LLM confidence-based thresholding, and a combination of these. This improved

DAG, dubbed DAG++, was shown to mitigate the issues with high-frequency APIs, while maintaining the accuracy of low-frequency APIs. Jain et al. also benchmark these methods with different models, showing an improvement on averages across all models.

Building on top of the work of Jain et al. and others, a dive into the usefulness of the RAG method by Chen et al. [41] focuses on less common API libraries. While coming to the same conclusion that RAG improves the accuracy of an LLM, their more notable contribution is the finding of the importance of documentation quality, especially the presence, diversity and accuracy of code examples. Other factors, such as lacking description were not as important to the RAG accuracy, while removing parts from the documentation interestingly had a positive impact on it, such as parameter lists. They also found that minor noise in the documentations was a non-issue, due to the excellent noise-tolerance of the LLMs.

The previous articles were a partial continuation of the work by Patil et al. in their Berkeley research [42] on training a context-aware LLM to be compared against other models, while using their Retriever-Aware Training technique (which is essentially RAG). Their fine-tuned context-aware Gorilla model outperformed all other LLMs on accuracy within the context in all test cases, and had zero or close to zero API hallucinations with RAG enabled.

Referenced in the survey by Fan et al. [30], Geng et al. [43] studied code summarization with LLMs by assessing their multi-intent comment generation capabilities and how to improve the effectiveness of LLMs in this task. They demonstrate that with some tweaking, such as few-shot learning and professional selection, the generated code summary is in-context and dis-

plays multi-intent use description for the code block. Implications on how this would scale to massive projects and code blocks with implicit context were not prevalent in this study.

Luo et al. developed [44] a repository-level code documentation generation tool *RepoAgent* utilizing an LLM. They proclaimed that previous attempts at this lacked in summarization, guidance and updates, where summaries overlooked the dependencies within the broader context, the documentation did not guide developers properly on code usage, and the documentation did not reflect code changes in a timely manner. Their process was to first create an Abstract Syntax Tree (AST) from the source files and their contents within a project. They identified two types of reference relationships, namely Caller and Callee. With this contextual information, they extracted all bi-directional reference relationships to form a Directed Acyclic Graph (DAG) which was fed to an LLM to be used for the generation. While their blind preference study suggest that the output of *RepoAgent* was preferred over human-authored documentation, their evaluation method did not assess the accuracy of the documentation. Luo et al. also note that AI generated documentation may still require human for review.

Macke and Doyle present the problem of documentation's effect on LLM capabilities in their 2024 study on the effect of code documentation for LLM performance [45]. They dive deep into what it means for code to be understood in this context, and state that predicting input-output pairs is a good indicator of understanding the code. As an example, given a function and an input, what is the output of the function for the input, or vice versa. Using this as a basis, they used LLM to generate unit tests based on code

with varying content. They found that random comments have a high negative impact on success, but LLMs are almost impartial to name mangling. Interestingly, tests on uncommented versus commented code revealed that comments do not significantly increase code understanding of the LLM.

3.3 Reflection

To help answer RQ2, a review of 11 separate articles about API documentation with a focus on the use of generative AI was conducted. From these articles, three general topics on how AI may be utilized in this context were identified, shown in Table 3.4. As stated by Fan et al. [30] the focus of AI usage in software engineering appears to be on the code generation side. Most articles about API documentation and AI also investigate how documentation can be utilized to improve code generation. This is not the only avenue for AI usage in this context, as shown by articles [39] [43] [44], and the slightly out-of-context article [37] due to not using AI. Still, most of these were also considering the generation side of AI usage, instead of using the language prowess of LLMs to find unique use cases, such as error detection methods similar to [37].

Topic	Method
Documentation generation	Full API documentation, Code summary, Initial structure
AI code generation improvement	RAG
Error detection	Detecting code vs. documentation discrepancies

Table 3.4: Generative AI Utilization within the Documentation Space

The survey showed promise in fully automated API documentation gen-

eration with AI, but it is still too early for wider adoption. The need for human intervention during this process, as mentioned for example in articles [44] [38], suggest a partial adoption could still be possible. As described in [36], the developers would then become the curators of the documentation rather than its writers. This focus shift in documentation creation could become problematic, as the content of AI generated documentation's mimicking, while not being, an accurate one could be considered correct by a slight oversight of the content and the disconnect of the source. As discussed in Chapter 2, documentation should always be correct, as it is the main source of information for using the library or service. This survey reveals that more research on the correctness of AI generated documentation is still needed.

An approach that would be more immediately applicable for the topic of documentation generation is using generative AI for summarizing code. Using fine-tuned models dedicated to understand code, a code summary generator could be utilized during development to describe for example what a block of code does, what is its purpose and potential use cases, what side-effects it may have, and where to not use it. However, this is not as simple as it may seem at first. The code blocks may have dependencies and implicit knowledge baked in, therefore, the model must be context-aware and have access to a wider range of knowledge beyond the block itself. This context awareness becomes extremely complicated when different versions of libraries and dependencies from multiple sources are considered. If this becomes an implementation avenue for RQ2, based on [45], a preprocessor should selectively remove all comments before generation to achieve as stable results as

possible.

Testing documentation much like unit testing code is the most interesting concept revealed by this survey, albeit none of the reviewed articles used LLMs for this task. There could be an avenue for future research here, or in other non-generative tasks using documentation. And, while all of the topics and their methodologies discovered in this survey are far from perfect, they could all be immediately utilized for API documentation maintainability and usability in a manual manner, with an emphasis on human oversight.

4 Automated Documentation Generation

Answering to RQ1, the following sections in this chapter will embark on designing the initial API documentation system used for the underlying web application project. The chapter starts by identifying what issues motivate the work on building such a system, followed by defining a set of objectives for it. Then, a closer look on the design itself is presented, while trying to explain and demonstrate how the system operates within the confines of the NDA.

4.1 Problem Space

During the web portal stack update, a lack of proper API documentation of the previous web portal stack highlighted the need for updating the process of its creation. From my own experience with the exception of a few, software developers are often lazy when it comes to writing formal documentation. Therefore, the key to its success is to have it integrated to the development process and be as streamlined as possible. This necessitates automation of

mundane tasks, such as building and deploying the documentation for easy access. With a standardized method of the site generation, the design could be scalable to include not only the web stack API, but other projects within the company.

The new stack has been split into three major repositories in expectance of more projects utilizing the stack. Without going into technical details due to the NDA, two of the repositories are used as scaffolding for the third, which can be regarded as the repository of business logic, the output of which is presented to users. In contrast to the other two, the third lacks any exposed API. Nevertheless, usage documentation for the developers is still a necessity due to the scale and complexity of the project. While not a part of the stack update, the backend of this portal also requires proper API documentation, thus already setting the amount of different documentation trees to four.

Adding to the complexity of the documentation generation and site management is the need to support documentation for multiple branches and tags, as some major features may be in development for weeks or even months. The documentation should still be readily available, but not mixed with a main branch documentation until completion. This allows for other updates on the mainline, such as hotfixes or other features, while using the correct API documentation for it. The documentation should also always be up to date, and reflect the latest changes made to the repository while providing decent amount of penetration and adhere to the guidelines discussed in Chapter 2, increasing DX through the lifespan of the projects.

These constraints, namely the multiple active repositories and their respective branches and versions, lead to a hypothesis about the design: the

documentation system must be kept apart from the projects themselves. This separation of concerns also increases cohesion in the projects, as the consumed projects do not have to care about the presentation of the documentation. To keep a minimal level of entry complexity to include a project to the documentation system, the system should contain automation to gather information from these repositories, in order to generate the final documentation. Therefore, a cross-repository automated API documentation generation design is proposed. Initial thoughts on the difficulties of designing such a system are:

1. Generation may be expensive in terms of infrastructure resources. How to keep it as lean as possible, so that it scales well-enough to allow including new projects into the system?

2. Storing metadata about the projects included in the system is a must. Where and how to store it?

3. Outdated and unused documentation will probably pile up and waste resources. How can they be identified and cleaned while not affecting documentation still in-use?

4.2 Content Creation

The web portal source code across all the repositories contains mostly TypeScript and JavaScript, utilizing a popular frontend framework as a tool set. This allows for a streamlined content creation for the documentation system, as the tooling to extract information from source code can be shared be-

tween the repositories. This extraction process should create data that can be ingested by the site generator. Choices for this intermediate data format were JSON, Markdown, HTML, or something custom. Due to its prevalence, ease of use, and support in many static site generation systems, Markdown were chosen. On top of automatically generated content, Markdown allows for manually written documentation as well. It was also deemed superior for manually writing software documentation, due the close resemblance of the final result while writing, and the rendering capabilities and resulting aesthetics of it.

The de facto standard for documenting code in JavaScript is to use JSdoc [46] annotations, while TSdoc [47] is a proposal to standardize documentation comments in TypeScript. These systems resemble each one another, with the most notable difference being the ability for TSdoc to automatically infer data type information from the source code. JavaScript, lacking types within code, requires documentation level type annotations for proper type reference. With these systems as the baseline for commenting the source, a JSdoc/TSdoc parser was used to generate the Markdown files from each source file. For the site generation automation, the resulting Markdown file content is mostly irrelevant, thus a more detailed description of how it gets written is omitted. The only constraint was that the files are error-free, ergo renderable. Snippet 3 shows an example error which resulted in a site generation failure, encountered during testing. This type of unclosed HTML tag error was the only type found to cause failures in generation.

Snippet 3 Example Error in a Markdown File
Markdown

```
Text containing an HTML tag that should be closed, but is not
    e.g.: <data>
```

4.3 Site Generation

Due to the fact that this site generation had to consider multiple projects, a decision to include the documentation outputs of every project close to the generator was made. This method allows for a static site generator to parse all files and create a stylized static HTML site to serve the documentations from. RQ1a posits that the generation should be effectively organized to allow scaling with multi-repository project structures. A static site generator can be made project agnostic, given the format of the content remains the same. As discussed in the previous section, Markdown was chosen as the intermediate format for the documentation. This necessitated choosing a static site generator which could consume Markdown files. For research purposes, the choice is rather irrelevant, and thus is not disclosed.

Two possible routes for managing version information was identified; either through annotations using special tags to represent changes that have been made in each iteration, or a complete separation of each version by organizing them into subfolders for the site generator to parse from. Considering RQ1b the second one was determined to be a better approach, as it does not exclude the first, while also offering the possibility to include the documentation for not only tagged versions, but different branches too. While there will be duplication, and thus increased time in generation and

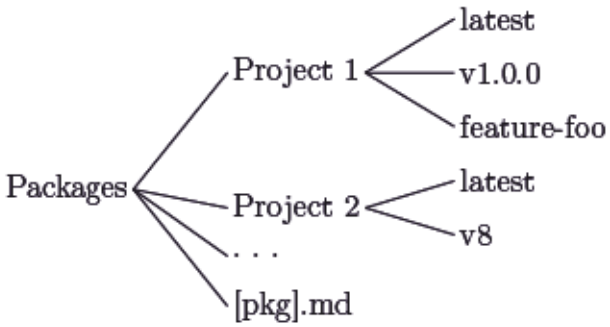

Figure 4.1: Site Generation Package Folder Structure

used resources, this allows for flexibility to select which versions to include in the final generated site. There is a downside to this approach in the form of searchability and the chosen static site generator's provided search algorithm, which we will discuss later with its implications for RQ1a. Figure 4.1 demonstrates how the folder layout is structured based on the information above.

To address RQ1a and to allow even more flexibility for the projects included in this system, the site generator had to be able to process different styles of project generated Markdown file structures. As seen in Figure 4.2, the structure of a project's Markdown files can be in a few different compositions. This was found during testing of the Markdown generators used with the projects mentioned earlier in this chapter, and rather than limiting to a single way to compose the documentation, all of the initially found compositions were accepted. Furthermore, the algorithm used for processing these compositions is customizable and replaceable when and if the need arises. Index.md was marked as a special file, which would be used as the documentation of its parent. For example, the v1.0.0/index.md file would be the main file for the whole version, while v1.0.0/Functions/Foo/Bar/index.md would

be the documentation for the Bar object. Paths, which contained a dot-symbol were split into separate paths, such that for example in the Figure 4.2, v1.0.0/Components/input.email.md would create a path for input and append an email documentation page for it. The algorithm for this process is depicted in Snippet 4 as pseudocode.

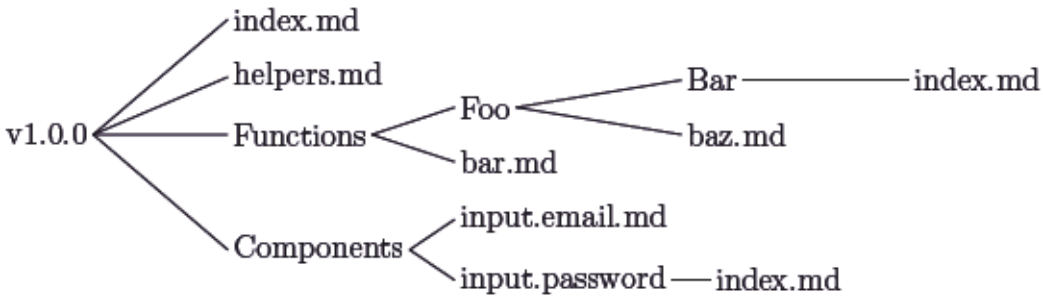

Figure 4.2: Example Markdown Folder Structure for a Single Version of a Project

Due to not being allowed to disclose the used technologies in this site generation, we are limited to discussing its capabilities, limitations and issues in abstract or functional terms. The searchability problem mentioned before can only be described but not shown in detail. We identified a potential scalability issue with it relating to RQ1a, as the filtering of results happens after the search, rather than before it. This approach is adequate when there are only some yet unknown amount of projects within the system, but could slow down drastically at some point when more content is included. The filtering should work, such that when browsing a specific version of a project, only the search results of that specific version should be show. On an upper level, for example at a project browse level, the search should include results from each version respectively, while the main page should include all

results. This was implemented as a custom filter based on the path of the current documentation page. A better approach to filtering in this scenario would be to omit all paths not relevant to the current path before the search begins. Due to current limitations in the provided search component, this method is not supported. This problem is left for future research, as it is yet unclear whether the indexing of the search algorithm is powerful enough to handle all of needs of this system.

To add more features for addressing RQ1a, the generation included a project browsing level mentioned earlier. The purpose of this is to get a quick glance about the higher level information of a project. It is generated along with a specially crafted Markdown file, which dynamically renders information about a project and its respective versions for each project. In Figure 4.1, the special files created for this are shown as [pkg].md, where the [pkg] gets replaced with the name of each package.

4.4 Deployment Automation

The documentation generated by the projects had to be stored somewhere. Due to opting for having multiple versions available simultaneously and the generator requiring them present at generation-time, the documentation trees had to be permanently stored somehow. Discussions for a solution to this problem were held in a design planning meeting, and a few alternative options were presented. In an attempt to answer RQ1a from the perspective of delivery using professional expertise and opinion, the solution which got selected was deemed the most streamlined in maintenance and a one that would scale

Snippet 4 Path Generation Algorithm for a Single Version

pseudocode

```
[RESULT] is an empty list of nested objects

# Find all Markdown files in a folder structure recursively
FOR EACH [FILE] IN files within [PATH] with .md ext:
    SET [REL] to path relative to [PATH], omit filename

    # [P1] and [P2] are pointers to objects
    POINT [P1] to [RESULT]
    POINT [P2] to NULL

    # Step 1. Create an object path within the [RESULT] for [REL]
    # [REL] can be an empty string in which case the following
    #   loop will not be iterated
    FOR EACH [SUBPATH] IN split [REL] with folder separator:

        # [SUBPATH] cannot be an empty string
        # [SUBPATH] may be in the form of foo, foo.bar or similar
        # split must return an array of n > 0
        FOR EACH [DP] IN split [SUBPATH] with ".":
            [P2] is FROM [P1] FIND elem where text equals [DP]
            unless ADD new elem to [P1] where text is [DP]
               and POINT [P2] to the new elem
            WHEN [P2] does not contain .items
               SET [P2.items] to new list
            POINT [P1] to [P2.items]

    # Step 2. Add more info to the [RESULT]
    IF [FILE.name] equals index.md
        # index.md at root does not have an object created for it
        #  therefore [P2] is NULL and a new object must be added
        IF [REL] is an empty string
            ADD new elem to [P1] WHERE:
                text is version interpreted from [PATH]
                link is [FILE.path]
        ELSE
            SET [P2.link] to [FILE.path]
    ELSE
        ADD new elem to [P1] WHERE:
            text is filename without ext
            link is [FILE.path]
```

the best at this stage of development. The options were as follows:

1. Pull documentation from a project into the documentation generator git repository with a CI/CD pipeline.

2. Push the documentation to the documentation generator git repository with a CI/CD pipeline from the project.

3. Fetch all project repositories during a site regeneration CI/CD pipeline and either pull from cache or rebuild all previous documentation based on metadata of which versions should be included.

4. Push the documentation to a host serving the generation from a CI/CD pipeline of a project

5. Use a webhook or similar method to notify a hosting service to pull the documentation as an artifact of a CI/CD pipeline of a project.

6. Poll the project git repositories for changes and pull the documentation when updated on a host serving the generation.

Storing documentation output inside a git repository was deemed unnecessary and excessive as there is no need to version control build outputs. Therefore, the first two options were dropped immediately. The third option was promising, but cache management or rebuilding all documentation multiple times did not seem feasible or a future-proof solution. This cache should be stored either within a runner's cache that does not expire, or build a separate cache to fetch from. Relying on such an unexpiring cache was not found to be a good practice, and from the separate cache system, there is a

short path to have all the necessary tooling within this system to generate and host the final site. Therefore, options 4 to 6 were considered.

From these three options, the first one was the selected solution. It does require initial setup for each project much like the other two, but has a more streamlined workflow and faster path to production. Permissions to write to the host had to be set up for each project to only allow for certain paths to be modified within the host. This had the added benefit of allowing each project to set up their versioning as necessary within the documentation system. Using a webhook to pull from the CI/CD pipeline's artifacts might be an upgrade to this, but requires the webhook system to be built. This is left for future design optimizations, as the push-to-host method was decent enough at this stage. Polling the git repositories was deemed to be an inferior method compared to the push-to-host due to the webhook possibility and the required scripting and setup needed to have it robust.

Figure 4.3 illustrates the workflow of this generation system as a UML sequence diagram, where *Docs Host* is the hosting service, which holds the documentation trees and a local copy of the git repository *git:docs-repo*, the site generation logic repository. *git:project-repo* is an example project git repository included in the documentation automation, and *git:ci/cd-runner* is a generic runner used for handling CI/CD requests made by the git repository.

The process starts with a developer pushing a local branch or a tag to a project repository. This triggers a pipeline to generate the documentation Markdown files for the specific branch or tag, which gets pushed to the host. The pipeline then triggers a downstream pipeline specified within the *git:docs-repo* repository, which sends a regeneration command to the host,

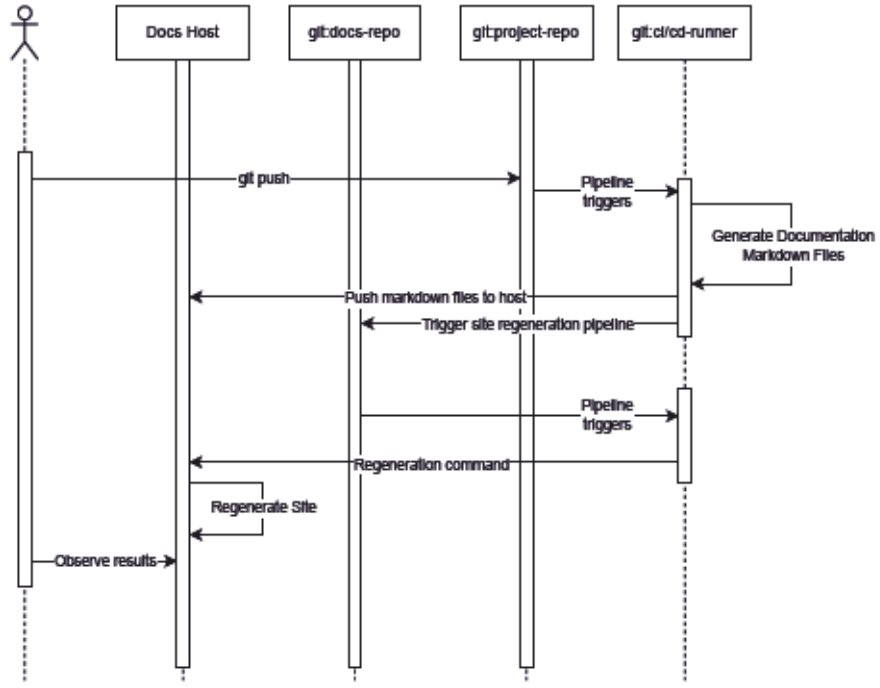

Figure 4.3: Documentation Generation and Hosting Workflow

starting a new site generation action. This trigger is specified within the *git:docs-repo* due to permission rules. Only this repository has the required permissions set up to run the generation within the hosting service. After these pipelines have all finished, the developer can observe the results from the hosting service, and optionally inspect the logs of the pipelines from the git platform.

Currently, the plan for the deletion of outdated and unused versions and packages from this system is to do it by hand. There are currently no requirements for its automation, as the system is still small and the manual effort is at an inconvenience level. Additionally, the static site generator selected to handle the site generation from Markdown has built-in caching,

reducing the amount of processing necessary should the amount of projects and versions pile up. With the caching enabled and working as intended, the primary issues arising from excess versions and packages should be limited to physical storage consumption and the aforementioned searchability issue.

5 Evaluation

This chapter evaluates the design at both subjective and objective levels. Metrics that can be pulled from the system are quite limited as of yet, thus the subjective part may overshadow the objective part in analysis. Firstly, a definition on how the system should be evaluated based on answering RQ1 is presented, followed by the objective and subjective evaluations. As the evaluation was not performed by an independent assessor, the potential for bias should be noted in the subjective evaluation.

From the perspective of automation, the evaluation metrics of interest for this system can be classified into the categories of scalability, usability and correctness. Scalability consists of performance questions, such as how long does build or generations last and do they increase with scaling, while usability assesses aspects such as the ease of deployment and branching to new projects. Usability from this perspective does not include the documentation's goodness metrics, such as searchability, penetrability, efficiency of reading, learnability, and so on. Correctness from an automation standpoint can be defined as the inclusion of all content within the final site which were passed along the pipeline from the source.

#	Project Number	Tree Count	Documentation Build Duration	Documentation Deployment Duration	Site Generation Duration
1	1	1	38 s	2 s	22 s
2	1	1	40 s	2 s	17 s
3	1	2	47 s	2 s	19 s
4	2	3	10 s	2 s	20 s
5	1	3	36 s	3 s	20 s
6	2	4	10 s	2 s	20 s

Table 5.1: Duration Metrics from Git Hosting Service CI/CD Pipelines

5.1 Objective Evaluation

To try and answer RQ1a and RQ1b objectively, run durations of the pipelines have been logged, while increasing the amount of projects and versions included in the system. Scalability concerns for the search function were not measured, as the indexing has remained fast enough that a human cannot notice a difference after including more documentation, and the issue cannot be fixed as it stands, as mentioned in the previous chapter. Correctness cannot be properly evaluated as it is not permitted to be demonstrated under the NDA. This would require displaying the source, its comments and the generated output, and verify whether the output matches the source.

Table 5.1 presents these results, from which observations about the system's scalability are not yet obvious. The site generation duration seems to remain close to constant as the tree increases. There is some overhead from orchestrating this with the CI/CD runners, but it is relevant to the final product and therefore included in these measurements. More data points should be collected to validate whether the caching of the static site generator and the efficiency of the scaling are adequate. From these numbers a definite answer for RQ1 cannot be deduced, but the efficiency of the site generation

with scaling is promising. The runner used in this test was running on a virtual machine, with a 2-core CPU and 8 GB of RAM. The hosting service was a similar virtual machine. All runs were done on the same machines. The columns in Table 5.1 from left to right are: index for the run, a number representing a project, total amount of documentation trees (a version of a project) for the site generation, time it took to build the documentation for the project, time it took to deploy the documentation to the host, and time it took to generate the final site. The numbers are taken from the git hosting platform's elapsed times of the pipeline runs.

5.2 Subjective Evaluation

Adding projects to this system is rather simple. The requirements are limited to an initial setup as follows; create permissions to write to the host, add a documentation generation system for the project, add a pipeline steps to push the generated documentation to the host, and trigger a downstream pipeline for the regeneration of the site. Most of these can be implemented by directly copy-pasting from existing implementations. If the permissions are created correctly, there is little room for misconfigurations which could end up breaking the system. A more glaring problem arises from generating content that is erroneous as discussed in the previous chapter. While this will not break the documentation service itself, it does affect the site generation by failing the generation. Another layer of added robustness for this kind of an error should be considered in future improvements.

Modifications on the site generation system can also be done separately

from the projects themselves, easing the alterations for the site itself. These modifications can be visual or functional in nature. The added functionality can be deployed independently from the included projects, while preserving the content of the projects already present. Examples of these alterations include for example changing colors or layouts, adding meta-information about the site or the projects, adding functionality to for example browse the site more easily, and so on. This is made possible since the static site generator used allows for deep customization of the final site. The site generator itself can also be replaced if necessary, as the content is in an intermediate and widely used format.

From a subjective standpoint, the RQ1 can be summarized such as: since the system demonstrated to support multiple projects and their multiple versions simultaneously, while maintaining a sufficient build and deployment times, it is effectively organized within the context of the underlying web portal rewrite project. There is also a high probability that it performs adequately when scaled to encompass other projects.

6 Conclusions

This thesis looked at the problem of API documentation automation organization in a cross-repository multi-project setup, while researching on how to include generative AI use within the API documentation space. There were two main objectives, creating and implementing a design for automating the documentation generation and delivery to be evaluated, and investigating how AI can help developers by using, manipulating or creating API documentation. From these objectives, two research questions were derived and answered. While the questions were more on subjective side, there is an objective truth underneath, which was briefly touched upon.

For answering the RQ1, the implemented design managed to scale effectively with a multi-repository project structure, while simultaneously supporting multiple versions of said projects under the limited testing currently made. The limited objective metrics gathered, and the subjective analysis both indicate that the system will perform well under higher usage and scaling, but a definite answer for it cannot be deduced yet.

AI research within the software space is concentrated on generating code, instead of documentation. Therefore, as an answer to RQ2, research gaps still exists for AI usage within this space. Whether AI research reaches a level

where all code gets written by LLMs, rendering API documentation irrelevant remains to be seen. Until then, using AI to enhance API documentation is an interesting topic that could be researched more. Current attempts are still naively optimistic, and advocate for human intervention, especially on the generative side.

6.1 Limitations

The NDA disallowed discussions of technical details more deeply. This makes evaluation of the design as it currently is impossible for other parties. Nevertheless, the design can be replicated closely with the description given in Chapter 4, while utilizing common open source tooling and professional expertise.

Objective measurements of the performance of the design were shallow as the system is still in its infancy. More data must be gathered from its usage to adequately assess its ability to scale. While mock-up tests could have been generated to test the system, it would not have been feasible within the timeframe to complete this thesis, due to constraints of the infrastructure and deadlines for the rewrite project itself. Therefore, the design had to be subjectively analysed. The system is in production internally within the company, and more data will be gathered.

6.2 Future Work

Whilst continuity of the design will be kept in-house, generative AI usage within the documentation space should pick up some insights gathered from the survey of Chapter 3. This will probably see a study path created for it within the company as well. The design allows for customization of the site, which is necessary to include the use of AI in it.

Figuring out implementation avenues outside of documentation generation with generative AI would be of great interest. One might argue that when an LLM can produce exact and accurate API documentation, it will be capable of perfectly using it to produce high-quality software, rendering the documentation generation pointless. This stems from the fact that proper documentation of an API requires a deep understanding of the underlying code, and when the code is well understood, there should be a short path to utilizing it more effectively.

References

[1] J. v. Brocke, A. Hevner, and A. Maedche, "Introduction to Design Science Research", in Sep. 2020, pp. 1–13, ISBN: 978-3-030-46780-7. DOI: 10.1007/978-3-030-46781-4_1.

[2] J. Y. Khan, M. Tawkat Islam Khondaker, G. Uddin, and A. Iqbal, "Automatic Detection of Five API Documentation Smells: Practitioners' Perspectives", in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, ISSN: 1534-5351, Mar. 2021, pp. 318–329. DOI: 10.1109/SANER50967.2021.00037. [Online]. Available: https://ieeexplore.ieee.org/document/9425926/ (visited on 05/03/2025).

[3] Y. Zhou, C. Wang, X. Yan, T. Chen, S. Panichella, and H. Gall, "Automatic Detection and Repair Recommendation of Directive Defects in Java API Documentation", *IEEE Transactions on Software Engineering*, vol. 46, no. 9, pp. 1004–1023, Sep. 2020, ISSN: 1939-3520. DOI: 10.1109/TSE.2018.2872971. [Online]. Available: https://ieeexplore.ieee.org/document/8478004 (visited on 05/20/2025).

[4] S. M. Sohan, C. Anslow, and F. Maurer, "SpyREST: Automated RESTful API Documentation Using an HTTP Proxy Server (N)", in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2015, pp. 271–276. DOI: 10.1109/ASE.2015.52. [Online]. Available: https://ieeexplore.ieee.org/document/7372015 (visited on 05/20/2025).

[5] M. Michalski, P. Kosko, D. Juszczak, and H. Kwon, "EWIDL: Single-Source Web API Documentation Management System", in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ISSN: 2576-3148, Sep. 2020, pp. 723–726. DOI: 10.1109/ICSME46990.2020.00081. [Online]. Available: https://ieeexplore.ieee.org/document/9240696 (visited on 05/20/2025).

[6] Y. Arimatsu, Y. Ishida, K. Noda, and T. Kobayashi, "Enriching API Documentation by Relevant API Methods Recommendation Based on Version History", in *2018 IEEE Third International Workshop on Dynamic Software Documentation (DySDoc3)*, Sep. 2018, pp. 15–16. DOI: 10.1109/DySDoc3.2018.00014. [Online]. Available: https://ieeexplore.ieee.org/document/8530115/ (visited on 05/03/2025).

[7] G. Uddin and M. P. Robillard, "How API Documentation Fails", *IEEE Software*, vol. 32, no. 4, pp. 68–75, Jul. 2015, ISSN: 1937-4194. DOI: 10.1109/MS.2014.80. [Online]. Available: https://ieeexplore.ieee.org/document/7140676/ (visited on 05/03/2025).

[8] M. Hosono, H. Washizaki, Y. Fukazawa, and K. Honda, "An Empirical Study on the Reliability of the Web API Document", in *2018 25th Asia-*

Pacific Software Engineering Conference (APSEC), ISSN: 2640-0715, Dec. 2018, pp. 715–716. DOI: 10.1109/APSEC.2018.00103. [Online]. Available: https://ieeexplore.ieee.org/document/8719480 (visited on 05/20/2025).

[9] A. Head, C. Sadowski, E. Murphy-Hill, and A. Knight, "When Not to Comment: Questions and Tradeoffs with API Documentation for C++ Projects", in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, ISSN: 1558-1225, May 2018, pp. 643–653. DOI: 10.1145/3180155.3180176. [Online]. Available: https://ieeexplore.ieee.org/document/8453133 (visited on 05/20/2025).

[10] M. Hosono, H. Washizaki, K. Honda, *et al.*, "Inappropriate Usage Examples in Web API Documentations", in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ISSN: 2576-3148, Sep. 2019, pp. 343–347. DOI: 10.1109/ICSME.2019.00052. [Online]. Available: https://ieeexplore.ieee.org/document/8919166 (visited on 05/20/2025).

[11] W. Maalej and M. P. Robillard, "Patterns of Knowledge in API Reference Documentation", *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1264–1282, Sep. 2013, ISSN: 1939-3520. DOI: 10.1109/TSE.2013.12. [Online]. Available: https://ieeexplore.ieee.org/document/6473801/ (visited on 05/03/2025).

[12] A. S. Alalaq, "The History of the Artificial Intelligence Revolution and the Nature of Generative AI Work", en, *DS Journal of Artificial Intelligence and Robotics*, vol. 2, no. 4, pp. 1–24, 2024, ISSN: 25839926.

DOI: 10.59232/AIR-V2I4P101. [Online]. Available: https://www.dsjournals.com/air/AIR-V2I4P101 (visited on 05/08/2025).

[13] "ISO/IEC/IEEE International Standard - Systems and software engineering – Design and development of information for users", *ISO/IEC/IEEE 26514:2022(E)*, pp. 1–76, Jan. 2022. DOI: 10.1109/IEEESTD.2022.9690115. [Online]. Available: https://ieeexplore.ieee.org/document/9690115 (visited on 05/25/2025).

[14] B. A. Myers, "Human-Centered Methods for Improving API Usability", in *2017 IEEE/ACM 1st International Workshop on API Usage and Evolution (WAPI)*, May 2017, pp. 2–2. DOI: 10.1109/WAPI.2017.2. [Online]. Available: https://ieeexplore.ieee.org/document/7965483 (visited on 05/20/2025).

[15] M. Meng, S. M. Steinhardt, and A. Schubert, "Optimizing API Documentation: Some Guidelines and Effects", en, in *Proceedings of the 38th ACM International Conference on Design of Communication*, Denton TX USA: ACM, Oct. 2020, pp. 1–11, ISBN: 978-1-4503-7525-2. DOI: 10.1145/3380851.3416759. [Online]. Available: https://dl.acm.org/doi/10.1145/3380851.3416759 (visited on 05/03/2025).

[16] M. P. Robillard and R. DeLine, "A field study of API learning obstacles", en, *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, Dec. 2011, Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 6 Publisher: Springer US, ISSN: 1573-7616. DOI: 10.1007/s10664-010-9150-8. [Online]. Available: https:

//link-springer-com.ezproxy.utu.fi/article/10.1007/s10664-010-9150-8 (visited on 05/24/2025).

[17] *Comparison of documentation generators*, en, Page Version ID: 1289658569, May 2025. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Comparison_of_documentation_generators&oldid=1289658569 (visited on 05/25/2025).

[18] *Go/src/bufio/bufio.go at master · golang/go*. [Online]. Available: https://github.com/golang/go/blob/master/src/bufio/bufio.go (visited on 05/25/2025).

[19] *Bufio package - bufio - Go Packages*. [Online]. Available: https://pkg.go.dev/bufio (visited on 05/25/2025).

[20] *What's a story? | Storybook docs*. [Online]. Available: https://storybook.js.org/docs/get-started/whats-a-story (visited on 06/04/2025).

[21] *Storybook: Frontend workshop for UI development*, en. [Online]. Available: https://storybook.js.org (visited on 05/25/2025).

[22] *Git*. [Online]. Available: https://git-scm.com/ (visited on 05/31/2025).

[23] *Beyond Git: The other version control systems developers use - Stack Overflow*, en, Jan. 2023. [Online]. Available: https://stackoverflow.blog/2023/01/09/beyond-git-the-other-version-control-systems-developers-use/ (visited on 05/31/2025).

[24] *GitHub Actions*, en, 2025. [Online]. Available: https://github.com/features/actions (visited on 05/31/2025).

[25] *What is a static site generator?*, en-us. [Online]. Available: https://www.cloudflare.com/learning/performance/static-site-generator/ (visited on 05/31/2025).

[26] *What is a Static Site Generator? How do I find the best one to use?*, en, Apr. 2020. [Online]. Available: https://www.netlify.com/blog/2020/04/14/what-is-a-static-site-generator-and-3-ways-to-find-the-best-one/ (visited on 05/31/2025).

[27] *6 biggest LLM challenges and possible solutions*, en, Feb. 2025. [Online]. Available: https://nexos.ai/blog/llm-challenges/ (visited on 05/31/2025).

[28] *Open Source Licenses: Types and Comparison*, en-US. [Online]. Available: https://snyk.io/articles/open-source-licenses/ (visited on 05/31/2025).

[29] *The GNU General Public License v3.0 - GNU Project - Free Software Foundation*. [Online]. Available: https://www.gnu.org/licenses/gpl-3.0.en.html (visited on 05/31/2025).

[30] A. Fan, B. Gokkaya, M. Harman, *et al.*, "Large Language Models for Software Engineering: Survey and Open Problems", in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, May 2023, pp. 31–53. DOI: 10.1109/ICSE-FoSE59343.2023.00008. [Online]. Available: https://ieeexplore.ieee.org/document/10449667 (visited on 05/10/2025).

[31] V. Murali, C. Maddila, I. Ahmad, *et al.*, *AI-assisted Code Authoring at Scale: Fine-tuning, deploying, and mixed methods evaluation*,

arXiv:2305.12050 [cs], Feb. 2024. DOI: 10.48550/arXiv.2305.12050. [Online]. Available: http://arxiv.org/abs/2305.12050 (visited on 05/31/2025).

[32] *IEEE Xplore*. [Online]. Available: https://ieeexplore.ieee.org/Xplore/home.jsp (visited on 05/17/2025).

[33] *ScienceDirect.com | Science, health and medical journals, full text articles and books*. [Online]. Available: https://www.sciencedirect.com/ (visited on 05/17/2025).

[34] *Scopus preview - Scopus - Welcome to Scopus*. [Online]. Available: https://www.scopus.com/ (visited on 05/17/2025).

[35] *ACM Digital Library*, en. [Online]. Available: https://dl.acm.org/ (visited on 05/17/2025).

[36] A. H. Duin, I. Pedersen, J. Hall, D. Card, and L.-A. K. Breuch, "Co-AI Technical Writing: Documentation, Experimentation, User Testing, & Ethical Design", in *2023 IEEE International Professional Communication Conference (ProComm)*, ISSN: 2158-1002, Jul. 2023, pp. 41–43. DOI: 10.1109/ProComm57838.2023.00006. [Online]. Available: https://ieeexplore.ieee.org/document/10229250 (visited on 05/10/2025).

[37] 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)*, ISSN: 2576-3148, Oct. 2023, pp. 553–557. DOI: 10.1109/ICSME58846.

2023.00071. [Online]. Available: https://ieeexplore.ieee.org/document/10336349 (visited on 05/10/2025).

[38] A. Al-Ahmad, H. Kahtan, L. Tahat, and T. Tahat, "Enhancing Software Engineering with AI: Key Insights from ChatGPT", in *2024 International Conference on Decision Aid Sciences and Applications (DASA)*, Dec. 2024, pp. 1–5. DOI: 10.1109/DASA63652.2024.10836262. [Online]. Available: https://ieeexplore.ieee.org/document/10836262 (visited on 05/10/2025).

[39] P. Dhyani, S. Nautiyal, A. Negi, S. Dhyani, and P. Chaudhary, "Automated API Docs Generator using Generative AI", in *2024 IEEE International Students' Conference on Electrical, Electronics and Computer Science (SCEECS)*, ISSN: 2688-0288, Feb. 2024, pp. 1–6. DOI: 10.1109/SCEECS61402.2024.10482119. [Online]. Available: https://ieeexplore.ieee.org/document/10482119 (visited on 05/10/2025).

[40] N. Jain, R. Kwiatkowski, B. Ray, M. K. Ramanathan, and V. Kumar, *On mitigating code LLM hallucinations with API documentation*, en, 2025. [Online]. Available: https://www.amazon.science/publications/on-mitigating-code-llm-hallucinations-with-api-documentation (visited on 05/12/2025).

[41] J. Chen, S. Chen, J. Cao, J. Shen, and S.-C. Cheung, *When LLMs Meet API Documentation: Can Retrieval Augmentation Aid Code Generation Just as It Helps Developers?*, arXiv:2503.15231 [cs] version: 1, Mar. 2025. DOI: 10.48550/arXiv.2503.15231. [Online]. Available: http://arxiv.org/abs/2503.15231 (visited on 05/12/2025).

[42] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez, "Gorilla: Large Language Model Connected with Massive APIs", en, Nov. 2024. [Online]. Available: https://openreview.net/forum?id=tBRNC6YemY (visited on 05/12/2025).

[43] M. Geng, S. Wang, D. Dong, *et al.*, *Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning*, arXiv:2304.11384 [cs], Jun. 2023. DOI: 10.48550/arXiv.2304.11384. [Online]. Available: http://arxiv.org/abs/2304.11384 (visited on 05/12/2025).

[44] Q. Luo, Y. Ye, S. Liang, *et al.*, "RepoAgent: An LLM-Powered Open-Source Framework for Repository-level Code Documentation Generation", 2024, pp. 436–464. DOI: 10.18653/v1/2024.emnlp-demo.46.

[45] W. Macke and M. Doyle, "Testing the Effect of Code Documentation on Large Language Model Code Understanding", 2024, pp. 1044–1050. DOI: 10.18653/v1/2024.findings-naacl.66.

[46] *Use JSDoc: Index*. [Online]. Available: https://jsdoc.app/ (visited on 05/03/2025).

[47] *What is TSDoc? | TSDoc*, en. [Online]. Available: https://tsdoc.org/ (visited on 05/20/2025).

Appendix A About the use of generative AI for this thesis

For this thesis, ChatGPT was used to aid in the following areas:

1. Literature review article exclusion. Only used during the exclusion process for the exclusion. All the articles were personally verified and read. Detailed process explained within the chapter.

2. Sentence improvement suggestions. Given an initial sentence or a sub-sentence, the AI was prompted to provide alternative sentence structure ideas. Only a handful of sentences per chapter were improved by AI. Instructions were as follows:

   > *Every question is about sentence improvements. The improvements should be in an academic tone by default. Provide a few alternative improvements, and some suggestions on drastic changes. The format doesn't have to be exactly like this, use your best judgement.*
   >
   > *The provided sentence is not always a full sentence. It may be partial, have tacit knowledge, or refer to previous sentences.*

Also, provide honest and clear feedback on the given sentence based on the information above. Do not provide feedback if there is no need for it. The sentence may also be fine, don't try to force it not fine.

Example prompt for Chapter 2: Often times during writing any software using a new API, me and my co-workers find ourselves complaining about bad API documentation, especially when it seems to be incorrect.

Response: This sentence conveys a valid observation, but the tone can be made more academic, the grammar slightly polished, and the structure clarified. Here are several improved versions:

Refined Academic Alternatives

- "While developing software with unfamiliar APIs, my colleagues and I frequently encounter and express frustration over poor or inaccurate documentation."
- "When working with new APIs, my co-workers and I often find ourselves frustrated by inadequate or misleading documentation."
- "Software development involving new APIs frequently leads my colleagues and me to criticize the quality and accuracy of their documentation."
- "It is common for my colleagues and me to encounter subpar or erroneous documentation when working with unfamiliar APIs, often leading to significant development friction."

Suggestions for More Drastic Changes

To focus more sharply on the systemic issue or to shift the tone toward analysis or reflection:

- "The prevalence of inaccurate or poorly structured documentation is a recurring challenge we face when engaging with unfamiliar APIs, often hindering productivity and increasing cognitive load."
- "Our frequent dissatisfaction with API documentation—especially when it is outdated or misleading—highlights a broader issue in the software development ecosystem."
- "The recurring frustration my team experiences with poorly documented APIs underscores the need for better standards in API documentation practices."

Let me know if you'd like to frame it more as a research motivation or problem statement.

3. LaTeX usage. No constraints.