Automated Code Documentation Generation using Langchain with Embeddings in GPT 3.5 Turbo

Finn Handmann[1†], Jimmy Neitzert[1†], Albert Makdisi[1†], Parinaz Sanati[1†]

[1]Institut Informatik, Hochschule Ruhr West, Lützowstraße 5, Bottrop, 46236, North Rhine-Westphalia, Germany.

[†]These authors contributed equally to this work.

This chapter delves into the use of embeddings within Large Language Models (LLMs) to enhance the process of generating code documentation. These embeddings are abstracted into high-dimensional vector spaces, enabling LLMs to capture intricate relationships between code elements and their contextual usage. This methodology aims to create well-organized and context-rich code documentation.

1 INTRODUCTION

In recent years, the field of generated text has experienced a transformative shift with the emergence of large language models (LLMs) like GPT-3. This publication explores a groundbreaking technique that leverages embeddings within LLMs to automatically generate code documentation, a task traditionally carried out by human developers.

This publication delves into the use of embeddings within LLMs to enhance the process of generating code documentation. These embeddings, abstracted into high-dimensional vector spaces, enable LLMs to capture intricate relationships between code elements and their contextual usage. The repository, which serves as input, gets chunked and embedded into the LLM. One of the key applications of this approach is the generation of code documentation for software repositories. A structured methodology for this process is presented in Chapter 3, involving repository retrieval, file

filtering, chunking, embedding, and sequential chapter generation. This methodology aims to create well-organized and context-rich code documentation, which can significantly enhance code comprehension and maintainability.

2 State of the Art

This chapter delves into the innovative technique of generating code documentation using embeddings within LLMs. A breakthrough exemplified by its successful implementation through in-line documentations, is showcased in the publication "Automatic Code Documentation Generation Using GPT-3" [1]. LLMs are built upon the transformer architecture, which comprises multi-head self-attention mechanisms and feed-forward neural networks. The architecture's self-attention mechanism allows the model to weigh the importance of different words in a sentence based on their contextual relevance. This mechanism enables LLMs to capture long-range dependencies and contextual nuances effectively. In this approach it is used to generate a documentation of a given repository.

2.1 Large Language Models

LLMs are used for a variety of tasks such as language translation, question answering, and text completion. They have the ability to generate coherent and fluent text that is often indistinguishable from human writing. This feature can be used to automatically generate written information which would normally be written by a human. The most known publisher is OpenAI with the GPT LLMs. Table 1 gives an overview about the currently published large LLMs of OpenAI. The amount of parameters show the complexity of the models, but correlates only to a limited extent with its performance. The performance of the mentioned LLMs increased with every iteration of the model (source university exam score). Additionally the token length of the models increased. In English, tokens commonly range in length from one character to one word [2]. The increase of the maximum token length gives the ability for more complex prompts and results. The training date can be relevant for the generated results. The corresponding model is only trained on data which exists before the training date. This gets partly compensated in the newer models GPT-3.5 Turbo and GPT-4, because of their ability to learn with Reinforcement Learning from Human Feedback (RLHF) [3]. This method constantly improves the models with human inputs and evaluations of the results.

Information	GPT-3	GPT-3.5 Turbo	GPT-4.0
Parameters	125-175 Billion	154 Billion	1 Trillion
Max Tokens	2k	4k-16k	32k
training date	10.2019	09.2021	09.2021
Cost		0.0015-0.004	0.03-0.12

Table 1 Comparison of OpenAI's large language models [4].

2.2 Functionally of Embeddings

Additionally the LLM can be trained with selected data to specify and enrich the context. This functionality is called embedding. Embeddings place similar words or clusters of words close to each other in the vector space and dissimilar ones farther apart. This property allows the model to understand relationships between words, such as synonyms, antonyms, and contextual associations. Words with similar meanings will have similar embedding vectors. An example is shown in Figure 1. These information can be used to generate context rich responses on a request to the LLM [5].

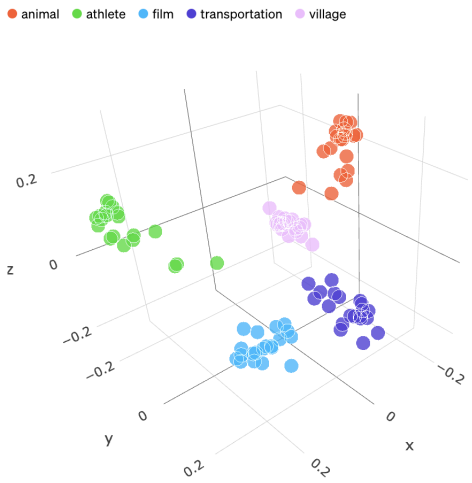

Fig. 1 Vector space of embedded data [5].

2.3 Langchain

Langchain, a novel architectural concept, extends the capabilities of traditional LLMs. Unlike conventional LLMs that process sequences in a left-to-right manner, Langchain incorporates bidirectional processing. This enables the network to consider future as well as past context, enhancing its ability to capture intricate patterns in sequential data. By allowing information to flow bidirectionally through different layers, Langchain can better model the underlying dependencies and relationships in the input sequence.

Langchain has the ability to optimize prompts of large language models. It can chunk a large amount of context into chunks and feed them into embeddings or a prompt. An overlap between chunks can increase the context between chunks, especially if code is chunked.

The combination of Langchain and embeddings enhances sequence modeling tasks. By chunking the context into small chunks embeddings can be trained more effectively [6]. With Langchain's bidirectional architecture, the model gains the ability to leverage both local and global contextual information. This synergy addresses challenges related

to context preservation, which can be critical for tasks like sentiment analysis, machine translation, and named entity recognition.

2.3.1 Conversational Retrieval Chain

A Conversational Retrieval Chain (CRC) is a category of conversational information retrieval (CIR) systems. It is designed as an information retrieval (IR) system that features a conversational interface. This interface allows users to engage in natural language conversations, whether spoken or written, to find information [7].

The key enhancement in the Conversational Retrieval QA(Question Answering) chain comes from its foundation on the RetrievalQAChain, with an added chat history component. This system takes the ongoing conversation history into account. It begins by combining the existing chat history, which is either given explicitly or retrieved from the system's memory, with the current question. This combination creates a new comprehensive question. The system then identifies relevant documents from its retriever based on this comprehensive question. Finally, the identified documents along with the comprehensive question are forwarded to a question answering chain. This chain processes the input and produces a response, which is then provided back to the user [8].

In the automated code documentation generator, the Conversational Retrieval Chain (CRC) is employed, while the Conversational Memory component of CRC is intentionally omitted to prevent information loss. Utilization of the memory function may potentially lead to an overload of information in the Large Language Model (LLM) conversation. Nonetheless, it remains indispensable for the incorporation of embeddings into the prompt.

2.4 Automatic Code Documentation Generation Using GPT-3

In the chapter "Automatic Code Documentation Generation Using GPT-3", the effectiveness of GPT-3 Codex, a Generative Pre-trained Transformer model developed by OpenAI specifically for code applications, in automated inline code documentation generation is evaluated [9]. The authors highlight the importance of well-written documentation in software development and the challenges associated with creating and maintaining it. They discuss existing approaches, such as template-based and information retrieval-based strategies, as well as learning-based methods using deep learning techniques [1].

The authors conducted a preliminary case study to evaluate Codex's performance in code documentation generation for six programming languages: Python, Java, PHP, JavaScript, Go, and Ruby. They used the CodeSearchNet dataset and compared Codex's performance with other existing models. The evaluation metric used was BLEU score, which measures the similarity between generated and reference documentation [1].

The results show that Codex with one-shot learning achieves state-of-the-art performance in inline code documentation generation, outperforming other models in terms of average BLEU score. It performs particularly well in Python, Ruby, JavaScript, and Go. For Java and PHP, Codex is the second-best performer, slightly

behind REDCODER and CodeBERT, respectively. The authors also conducted qualitative analyses of the generated documentations, considering factors such as readability, quantity, and informativeness. They found that Codex's generated documentations are comparable to actual ones in terms of these metrics and may even contain more comprehensible information in some cases [1].

In conclusion, this publication presents a systematic evaluation of the GPT-3 Codex model in automated in-line documentation generation. The results demonstrate the model's effectiveness, especially when provided with one-shot learning. The authors suggest that Codex has the potential to be a valuable tool for developers in generating code documentation and improving software development efficiency.

A subset of the repositories employed in this study serves as the basis for assessing the performance of the code documentation generator introduced in this chapter. The repository dataset is expanded by incorporating manually selected repositories created after the training date of the GPT-3.5 and GPT-4 LLM.

2.5 Guidelines for Effective Code Documentation

To evaluate the quality of code documentation, certain criteria need to be established. These criteria ensure that the documentation provides valuable insights and guidance for users and developers.

To assess the quality of code documentation, it's essential to establish clear evaluation criteria. Effective code documentation ensures that users and developers can readily understand and utilize the codebase. The following guidelines serve as benchmarks for achieving comprehensive and well-structured documentation. The Documentation should encompass all crucial aspects. The installation instructions must provide comprehensive steps and relevant URLs for external tools. The usage section should include illustrative examples. Dependencies, whether standalone or part of setup, should list all necessary requirements. The Document has to include all functions. Each function's name, purpose, usage, parameters, return values, and exceptions must be described. For projects not acting as libraries or frameworks, the functions section can be omitted. Additionally it should include examples illustrating the program's functionality and usage. The Document needs to be structured with headings for section division, bullet points for listing items, and formatting for easy scanning. For increased structure, the document should use markdown as formatting guidelines. Headings, lists and code have to be formatted appropriately. The overall structure should facilitate accessibility. Each guideline contributes to a cohesive presentation, aiding both users and developers [10].

3 Architecture

In this section, the architecture designed for the generation of code documentation from a software repository is shown (Figure 2). The process involves several key steps. First, the target software repository is retrieved from the provided link through a download process that captures the repository's current state. Once obtained, relevant files required for code documentation generation are filtered to ensure that only

the necessary codebase components are considered for further processing. These filtered files are then divided into smaller, manageable units known as "chunks" to enhance embedding and enable a more precise approach to documentation generation. In the next step all chunks get embedded into the LLMs. After the generation of the embeddings, the actual process of generating code documentation begins. The documentation is organized into chapters, with each chapter corresponding to a specific explanation of the repository. Importantly, the generated documentation chapters are created sequentially, to increase the accuracy and level of detail for each chapter. The

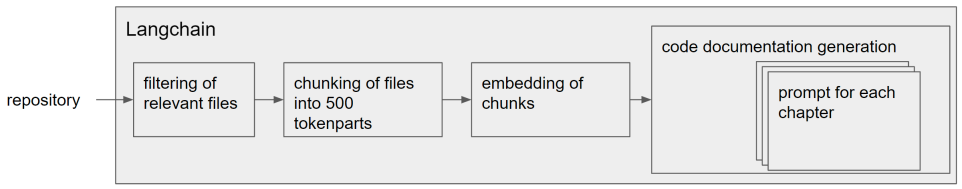

Fig. 2 Architecture of the code documentation generator tool.

architecture presented here demonstrates a structured methodology for code documentation generation, leveraging repository retrieval, file filtering, chunking, embedding, and a sequential chapter generation approach. This methodology enables the creation of well-organized and context-rich code documentation, which has the potential to significantly enhance code comprehension and maintainability.

3.1 Langchain and Embeddings

In this approach embeddings are used to generate the code documentation. The embeddings are generated with the help of Langchain. Langchain serves the purpose of processing the repository and extracting only the relevant textual content. For instance, it selects files with extensions like .py, .js, .c, and .go, and considers the content of the readme.md file. The code files are further divided into smaller segments, each containing 500 tokens with a 10-token overlap between each chunk. The chunks are also called "context windows". This overlap aids in creating context between chunks in the embeddings. The size of each chunk plays a crucial role in determining the efficacy of embeddings. If the chunks are too large, effective chunking does not occur as multiple contextual topics are amalgamated into a single chunk. Conversely, if the size is too small, insufficient contextual information is encapsulated within a single chunk. Optimal selection of chunk size enables the embeddings to categorize chunks effectively within its vector space, ultimately resulting in precise chunk selection and, consequently, more accurate results. The ultimate objective of this approach is to avoid embedding unnecessary files, thereby refining the quality of responses while also conserving tokens. With the embedding feature an analogy is drawn between the input and the embeddings as a heavily weighted vectorspace. Prompts which are used with embeddings use the most fitting embedded chunks to create the prompt answer the amount of chunks used can be configured for each prompt.

3.2 Code Documentation Generation

This section delves into the process of generating a code documentation using the pre-trained embeddings obtained through the Langchain. The outlined steps encompass loading the embeddings from the vector database and effectively utilizing the embeddings to streamline the documentation generation process. GPT-3.5 Turbo is used to increase the input amount of maximum embedding chunks for each prompt. Ths leads to results with a higher quality compaired to the more advanced GPT-4 LLM with less embedding chunks for each prompt.

The initial step is to read the embeddings from the vector database, which were generated using the previously outlined procedure. These embeddings encapsulate the code and information about the repository. All chapters get generated separately in individual chats and get merged to one code documentation file. The prompts for each individual chapter are listed in Table 2. It is important to add "only containing ..." to each prompt to avoid the generation of a whole code documentation in one prompt. Additionally the chance of duplication of content between chapters have to be taken into account. As such, each prompt must offer concise guidance on the chapter's content, ensuring a coherent and non-repetitive documentation outcome. The large language model lacks awareness of content covered in prior chapters, because every chapter is created in a new chat context. The amount of chunks from the embeddings for each prompt response is set to 35 chunks. The amount of chunks controls the complexity of the context and is limitted by the maximum token length of the LLM. Finally the code documentation is compiled by merging all the chapters listed in Table 2. The code documentation is formatted in markdown to easily implement it into the repository. The execution of prompts is facilitated through the utilization of a Langchain Conversational Retrieval Chain (CRC), to enable the integration of previously generated embeddings.

No.	Prompt
1	"Create a code documentation for the project,ONLY containing a brief summary of the project. Use the headline \"\#Summary\" and format it in markdown"
2	"Create a code documentation for the project,ONLY containing the dependecies of the project. Use the headline \"\#Dependencies\" and format it in markdown"
3	"Create a code documentation for the project,ONLY containing the setup instructions of the project. Use the headline \"\#Setup\" and format it in markdown"
4	"Create a code documentation for the project,ONLY containing the Installation instructions of the project. Use the headline \"\#Installation\" and format it in markdown"
5	"Create a code documentation for the project,ONLY containing code examples for the project (if applicable). Use the headline \"\#Examples\" and format it in markdown"
6	"Create a code documentation for the project,ONLY containing the usage instructions of the project. Use the headline \"\#Usage\" and format it in markdown"
7	"Create a code documentation for the project,ONLY containing the a documentation for each function. Ignore functions used for Testing frameworks. Use the headline \"\#Functions\" and format it in markdown"

Table 2 Code documentation generation prompts.

No.	Criteria
0	Completeness
1	Functions
2	Length
3	Language and Grammar
4	Examples / Usage
5	Readability
6	Formatting
7	Structure
8	Useless Information

Table 3 Code documentation evaluation criteria.

4 Evaluation

To comprehensively assess the effectiveness of the approach, an evaluation of the quality and performance of generated code documentation with a scoring is performed. By checking the performance of multiple generated code documentations and providing prove of significance a benchmark for its performance can be determined. The architecture of the evaluation is shown in Figure 3.

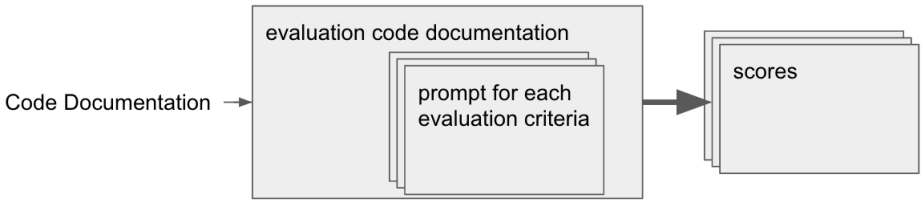

Fig. 3 Architecture of the code documentation evaluation tool.

4.1 Evaluation Strategy

In this evaluation strategy, we use the GPT 3.5 Language Model (LLM) to assess code documentation based on the criteria outlined in the predefined guideline. The guideline is based on the Google developer documentation style guide [10]. By using the advanced natural language processing capabilities of GPT 3.5, an evaluation of the the quality of code documentations is done by assigning scores to each evaluated documentation to make them comparable. This strategy creates a comprehensive, standardized and automated assessment of any given code documentation. The evaluation process evaluates the evaluation criteria, given in Table 3 in separate prompts to increase the accuracy of the result. This method can be used to evaluate the generated code documentation as well as already existing code documentations. This gives the ability to not only check the performance of the code documentation generator but also to compare it to already existing documentations.
The method to use a LLM for evaluation was already used to evaluate the performance of the vicuna LLM [11]. This is a reliable evaluation strategy for context evaluation,

if the evaluation criteria are described in detail and the prompt is designed properly. It is impotent to note that the results have to be checked. The results are based on stochastic probabilities. The probability for a meaningful result can be increased by fine tuning the prompt for evaluation and the evaluation criteria.

4.1.1 Usage of GPT-3.5 as Evaluator

GPT-3.5 Turbo has been chosen as an evaluator because of its available maximum token size of 16k. The given code documentation gets evaluated with a scoring of the categories given in Table 3. The quality of the code documentation is measured in a score from 0-1 with a resolution of one percent. It is important to configure the OpenAI API correct to get meaningful results. The goal is to get evaluation results which generates the same score for one document every time generated. This can be achieved by lowering the temperature of the GPT-3.5 Turbo model. The temperature describes the creativity in the result, which leads to a higher variation in the score. The best fitted temperature is a temperature level of 0.2 for our application. All chapters get evaluated separately in individual conversations. This reduces the complexity of each conversation, resulting in a more accurate outcome. The prompt for each evaluation criteria is listed in Table 4.

The primary determinant of achieving accurate and deterministic evaluation results resides in the careful design of the prompt. Merely presenting evaluation criteria as outlined in Table 3 is insufficient. Evaluation criteria must be elucidated and if needed enhanced with illustrative examples, facilitating comprehensive assessments across multiple code documentations. Furthermore, elucidating specific aspects of the criteria may warrant their weighting as "important" or "potentially misleading". The evaluation results necessitate examination. Should explanations and examples fail to yield the expected scoring outcomes, scoring guidelines may be introduced within the criteria structure, such as "0.2: Major problems -noticeable duplicates, language errors", or "0.4: Moderate - some duplicates, language issues affecting clarity".

4.1.2 Significance Tests

In order to ensure the reliability of the evaluation results, a significance testing to validate the statistical significance of observed differences between the quality scores of the generated code documentation and the pre-existing human-authored documentation is needed. This approach gives confident assertions about the effectiveness of the method in comparison to the established benchmarks. The significance testing framework enhances the robustness of the evaluation, providing a solid foundation for drawing meaningful conclusions from the results.

In order to determine a statistically significant difference between the evaluations of the original and generated code documentations, the statistical analysis tool SPSS was selected. Initially, the corresponding evaluation data, which includes the criteria listed in table 3 along with the average value for each tested GitHub repository, were tested for normal distribution. The results of the Kolmogorow-Smirnow-test and Shapiro-Wilk-test that were conducted in SPSS, indicated that the evaluation data was not normally distributed as could be seen in Figure 4. Nevertheless the t-test is still suitable as Pagano pointed out that the t-test has proved relatively robust

No.	Prompt
1	*Completeness*: The documentation should contain all relevant headings and the associated text should be exhaustive but concise. For example, the installation paragraph should contain all relevant steps to install the program and if applicable there should be an example in the usage paragraph, however if the installation paragraph skips some steps or no urls for external tools are used are considered misleading. If applicable a dependencies paragraph should be present (standalone or in the setup paragraph) and should contain all relevant dependencies.
2	*Functions*: Relevant functions should be documented with the function name as a heading and a description of the function's purpose and usage. The description should include any relevant parameters, return values, and exceptions. However if the project is not a library or a framework the functions paragraph can be omitted.
3	*Length*: The different paragraphs should be concise and to the point. For example, the installation paragraph should not contain a long description of the project, but rather a short description of the steps needed to install the program. Too long paragraphs are considered misleading. However, too short paragraphs are also considered misleading if there are infos missing.
4	*Language and Grammar*: Second person should be used (you rather than we), Standard American spelling should be used (color rather than colour), and the text should be free of grammatical errors. Conditions should be put before instructions (if you want to do X, do Y rather than do Y if you want to do X).
5	*Examples / Usage*: The documentation should contain at least one example of how to use the program. The example should be clear and easy to understand, and it should demonstrate the program's functionality. If the program is intended for use by other developers, the example should be written in a way that makes it easy to copy and paste into their own code.
6	*Readability*: The documentation should be written in a clear and concise manner, with no spelling or grammatical errors. The text should be formatted in a way that makes it easy to scan and skim through the document. If there are parts of the documents that are duplicate, consider these as VERY confusing (bad rating).
7	*Formatting*: The documentation should adhere to the criteria listed in the following examples. Use numbered lists for sequences. Use description lists for pairs of related pieces of data, and bullet points should be used to list items. Serial commas should be used (series of three or more items). Code in text or code samples e.g. in examples should be formatted as code blocks if the code consists of multiple lines otherwise backticks should be used.
8	*Structure*: Precisely assess the markdown-formatted documentation's structural organization, flow, and navigational aids on a scale from 0.0 to 1.0. Take into account the presence and clarity of section titles, logical content sequencing, consistent use of headings and subheadings, utilization of code blocks for examples, and the incorporation of navigation elements like hyperlinked table of contents. This will contribute to a more accurate and detailed rating.Precisely assess the markdown-formatted documentation's structural organization, flow, and navigational aids on a scale from 0.0 to 1.0. Take into account the presence and clarity of section titles, logical content sequencing, consistent use of headings and subheadings, utilization of code blocks for examples, and the incorporation of navigation elements like hyperlinked table of contents. This will contribute to a more accurate and detailed rating.
9	*Useless Information*: 0.0: Severe issues - extensive duplicate info, wrong language use. — 0.2: Major problems - noticeable duplicates, language errors. — 0.4: Moderate - some duplicates, language issues affecting clarity. — 0.6: Minor - limited duplicates, minor language problems. — 0.8: Slight - very few duplicates, minor language slip-ups. — 1.0: Excellent - no duplicates, accurate language use. / Guidelines: Evaluate duplicates, (programming) language correctness. Consider clarity, user guidance, consistency. Assess error identification accuracy. Adapt to project context. Feedback on specific issues essential.

Table 4 Code documentation evaluation prompts.

Tests of Normality

	Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Statistic	df	Sig.	Statistic	df	Sig.
AVG_A	,103	53	,200*	,935	53	,007
AVG_B	,216	53	<,001	,725	53	<,001
a0	,441	53	<,001	,615	53	<,001
b0	,468	53	<,001	,475	53	<,001
a1	,255	53	<,001	,835	53	<,001
b1	,422	53	<,001	,653	53	<,001
a2	,525	53	<,001	,345	53	<,001
b2	,360	53	<,001	,685	53	<,001
a3	,316	53	<,001	,607	53	<,001
b3	,444	53	<,001	,514	53	<,001
a4	,505	53	<,001	,395	53	<,001
b4	,284	53	<,001	,715	53	<,001
a5	,486	53	<,001	,494	53	<,001
b5	,423	53	<,001	,458	53	<,001
a6	,460	53	<,001	,587	53	<,001
b6	,230	53	<,001	,821	53	<,001
a7	,335	53	<,001	,761	53	<,001
b7	,384	53	<,001	,601	53	<,001
a8	,389	53	<,001	,699	53	<,001
b8	,407	53	<,001	,648	53	<,001

*. This is a lower bound of the true significance.

a. Lilliefors Significance Correction

Fig. 4 Results of tests of normality conducted in SPSS.

towards a violation of the normal distribution assumption [12]. Furthermore the sample size fulfills the condition of being "reasonably large" with 53 samples for each criteria and separate group-pairings as $n > 30$ [13]. The t-test compares the means of the samples using hypothesis testing to assess statistical significance. As the data consists of dependent samples (comparing evaluation values of AI-generated documentation and evaluation values for human-generated documentation), the paired sample t-test was selected and conducted in SPSS.

In order to prove the presence of a statistically significant difference between the evaluation values, the null hypothesis $H0$: "There is no difference between the evaluation values from the LLM generated code documentations and the original human code documentations." must be rejected in favor of the alternative hypothesis $H1$: "The evaluation values for the LLM generated code documentations are significantly higher than the evaluation values of the original human code documentations." The null hypothesis is rejected at the significance level with $\alpha = 0.05$ if $p < \alpha$. In this case, if $p < 0.05$, the null hypothesis $H0$ is rejected, and $H1$ has to be accepted.

4.2 Results

The code documentation generation is tested with 53 repositories. All 53 repositories have already existing code documentations, which are used as reference. 48 of 53 generated code documentations performed equal or better compaired its existing code documentation. Results are shown in detail in Table 7 in the attachments. The average score of generated code documentations is 0.78 with a standard deviation of 0.057 compared to 0.64 for existing documentations with a standard deviation of 0.204. The standard deviation of 0.057 of the generated documentations show that the quality between documentations is way more consistent compared to already existing documentations with standard deviation of 0.204. Table 5 displays the mean scores and standard deviations for each criteria. "a" describes the generated code documentations and "b" the existing code documentations. On a more detailed level the criteria "Functions" with an average of 0.483 and "Length" with 0.479 have the worst scores. Nonetheless the generated code documentation is outperforming the existing code documentation in the criteria "Functions" by a factor of 2. This could indicate a to strict evaluation for the critera. The "Length" critera is equaly rated between the existing and the generated code documentations, which results in a compairable quality in text length. The results show that the code documentation generator is a valid tool to automatically generate code documentations. It safes time and leads on average to a documentation with a higher quality.

Average Ø		Completeness Ø		Functions Ø		Length Ø		Language Grammar Ø		Examples Usage Ø	
A	B	a0	b0	a1	b1	a2	b2	a3	b3	a4	b4
0,778	0,636	0,877	0,764	0,483	0,222	0,479	0,521	0,830	0,766	0,968	0,615

Readability Ø		Formatting Ø		Structure Ø		Useless Information Ø	
a5	b5	a6	b6	a7	b7	a8	b8
0,911	0,842	0,770	0,549	0,915	0,802	0,766	0,645

Average σ		Completeness σ		Functions σ		Length σ		Language Grammar σ		Examples Usage σ	
A	B	a0	b0	a1	b1	a2	b2	a3	b3	a4	b4
0,057	0,204	0,047	0,230	0,347	0,340	0,082	0,182	0,067	0,235	0,092	0,443

Readability σ		Formatting σ		Structure σ		Useless Information σ	
a5	b5	a6	b6	a7	b7	a8	b8
0,038	0,248	0,107	0,312	0,060	0,252	0,174	0,282

Table 5 Average and standard deviation values for code documentation evaluation criteria.

4.2.1 Significance Tests

To interpret the results correctly, it must be clarified what the pairs in figure 5 are meant to represent. As previously explained, the evaluation results can be subdivided into different criteria, as shown in Table 3. Depending on the specified number, the

Paired Samples Test

		Paired Differences					t	df	Significance	
					95% Confidence Interval of the Difference					
		Mean	Std. Deviation	Std. Error Mean	Lower	Upper			One-Sided p	Two-Sided p
Pair 1	AVG_A - AVG_B	,14156184486	,20556419064	,02823641315	,08490133662	,19822235311	5,013	52	<,001	<,001
Pair 2	a0 - b0	,1132	,2378	,0327	,0477	,1788	3,466	52	<,001	,001
Pair 3	a1 - b1	,260849	,449345	,061722	,136994	,384704	4,226	52	<,001	<,001
Pair 4	a2 - b2	-,0415	,2089	,0287	-,0991	,0161	-1,447	52	,077	,154
Pair 5	a3 - b3	,0642	,2403	,0330	-,0021	,1304	1,944	52	,029	,057
Pair 6	a4 - b4	,3528	,4313	,0592	,2340	,4717	5,956	52	<,001	<,001
Pair 7	a5 - b5	,0698	,2423	,0333	,0030	,1366	2,098	52	,020	,041
Pair 8	a6 - b6	,22075	,30484	,04187	,13673	,30478	5,272	52	<,001	<,001
Pair 9	a7 - b7	,1132	,2557	,0351	,0427	,1837	3,223	52	,001	,002
Pair 10	a8 - b8	,1208	,3313	,0455	,0294	,2121	2,654	52	,005	,011

Fig. 5 Results of the t-test conducted in SPSS.

evaluation results for a criterion are grouped as numbered in Table 3. The letters "a" or "b" indicate the association with the evaluation results of the original or AI-generated documentations. While the AI-generated evaluation results are classified by the letter "a", the evaluation results of the original documentations are classified by "b". Only the significance values p of pair four and five, which represent the evaluation results for the criteria "Language and Grammar" and "Examples / Usage", accept the null hypothesis. Therefore the evaluation results for the KI-generated documentations and original human documentations do not show a significant difference regarding the mentioned criteria. The evaluation results for the remaining criteria like "Completeness", "Functions", "Length", "Readability", "Formatting", "Structure" and "Useless Information" do show a significant difference between the original and AI-generated documentations. As can be seen in figure 2, $p < \alpha; \alpha = 0,05$ for the mentioned criteria indicating a significant difference between the pairs. Since the alternative hypothesis H1 is a directed hypothesis, it can be said that the evaluation results for the KI-generated documentations are significantly better than the original documentations for the last specified criteria. Furthermore the average evaluation values, which form a mean for the specified criteria from table 3 for each repository documentation, show a statistically significant difference as $p < \alpha; \alpha = 0,05$.

5 Summary and Conclusion

In conclusion, the project examined the feasibility of utilizing LLM GPT-3.5 Turbo for the automatic generation of code documentation. The evaluation results demonstrated a significant statistical difference in favor of the LLM-generated documentations over the original existing documentations. This shows that the results of generated code documentations are not only more time efficient but also on average of higher quality. To achieve these results, several strategies were employed to optimize the utilization of GPT in the documentation process:

- Embedded files were segmented into appropriately sized chunks of approximately 500 tokens to facilitate effective grouping within the vector space, preserving contextual richness. This results in a better performance in the generated results using embeddings.

- Including the readme file in the embedding process provided deeper context and clarity to the generated code documentations, enhancing its overall quality.
- It's essential to chunk prompts into smaller, more manageable tasks. Only one criteria per prompt improved the consistency and accuracy of the generated code documentation. This approach helps in generating more detailed and consistent chapter of the code documentation.
- Initiating a new chat in every prompt proved beneficial in clearly separating different chapters of the code documentation, maintaining a structured and coherent narrative.

In conclusion, GPT demonstrates significant potential as a valuable asset in code documentation endeavors, offering substantial time and resource savings while outperforming existing documentation, as demonstrated by the statistical analysis. Nonetheless, generated documentation should undergo thorough review and validation before release to ensure alignment with established standards of clarity, accuracy, and completeness for the project. However, it's important to acknowledge that long, handwritten documentations have their own merits. They often provide a deep level of context and understanding that can be difficult to replicate with automated tools alone.

References

[1] Junaed Younus Khan and Gias Uddin. "Automatic Code Documentation Generation Using GPT-3". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE '22. Rochester, MI, USA: Association for Computing Machinery, 2023. ISBN: 9781450394758. DOI: 10.1145/3551349.3559548. URL: https://doi-org.hs-ruhrwest.idm.oclc.org/10.1145/3551349.3559548.

[2] OpenAI. *Tokenizer*. Sept. 2023. URL: https://platform.openai.com/tokenizer.

[3] OpenAI. *OpenAI RLHF*. Sept. 2023. URL: https://openai.com/research/instruction-following.

[4] OpenAI. *Pricing of GPT models*. June 2023. URL: https://openai.com/pricing.

[5] OpenAI. *Introducing text and code embeddings*. Sept. 2023. URL: https://openai.com/blog/introducing-text-and-code-embeddings.

[6] Pinecone. *Chunking Strategies for LLM Applications*. Sept. 2023. URL: https://www.pinecone.io/learn/chunking-strategies/.

[7] Jianfeng Gao et al. *Neural Approaches to Conversational Information Retrieval*. 2022. arXiv: 2201.05176 [cs.IR].

[8] Langchain Inc. *Store and reference chat history*. Aug. 2023. URL: https://python.langchain.com/docs/use_cases/question_answering/how_to/chat_vector_db.

[9] OpenAI. *OpenAI Codex*. Sept. 2023. URL: https://openai.com/blog/openai-codex.

[10] Google for Developers. *Google developer documentation style guide*. Sept. 2023. URL: https://developers.google.com/style.

[11] Wei-Lin Chiang et al. *Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality*. Mar. 2023. URL: https://lmsys.org/blog/2023-03-30-vicuna/.
[12] Robert Pagano. *Understanding statistics in the behavioral sciences*. Cengage Learning, 2012, p. 483.
[13] Neil J Salkind. *Encyclopedia of research design*. Sage, 2010, pp. 1560–1565.

6 Attachment

Repository	Average		Completeness		Functions		Length	
	A	B	a0	b0	a1	b1	a2	b2
1	0,78	0,60	0,90	0,80	0,50	0,00	0,50	0,80
2	0,78	0,58	0,80	0,80	0,80	0,50	0,50	0,50
3	0,80	0,42	0,90	0,80	0,80	0,00	0,50	0,50
4	0,79	0,73	0,90	0,90	0,50	0,00	0,50	0,50
5	0,74	0,74	1,00	0,80	0,00	0,80	0,50	0,50
6	0,71	0,67	0,80	0,80	0,50	0,00	0,50	0,50
7	0,86	0,68	0,90	0,80	1,00	0,50	0,50	0,50
8	0,73	0,70	0,90	0,80	0,00	0,00	0,50	0,40
9	0,71	0,71	0,90	0,90	0,00	0,00	0,20	0,50
10	0,76	0,74	0,90	0,90	0,50	0,00	0,20	0,50
11	0,76	0,75	0,80	0,80	0,50	0,50	0,50	0,50
12	0,83	0,73	0,90	0,80	0,80	0,00	0,50	0,50
13	0,86	0,57	0,90	0,80	1,00	0,00	0,50	0,50
14	0,71	0,79	0,80	0,90	0,50	0,80	0,50	0,50
15	0,73	0,72	0,90	0,90	0,00	0,00	0,50	0,80
16	0,72	0,73	0,80	0,80	0,00	0,00	0,50	0,50
17	0,77	0,74	0,90	0,90	0,50	0,50	0,50	0,50
18	0,79	0,56	0,90	0,90	0,50	0,00	0,50	0,50
19	0,77	0,60	0,90	0,90	0,50	0,00	0,50	0,50
20	0,86	0,72	0,90	0,90	1,00	0,00	0,50	0,80
21	0,80	0,76	0,90	0,80	0,50	0,50	0,50	0,50
22	0,84	0,80	0,90	0,90	1,00	0,88	0,50	0,50
23	0,86	0,76	0,90	0,80	1,00	0,50	0,50	0,50
24	0,79	0,63	0,90	0,00	0,50	0,00	0,50	0,00
25	0,77	0,76	0,90	0,80	0,00	0,00	0,50	0,50
26	0,78	0,73	0,80	0,80	0,50	0,80	0,50	0,50
27	0,81	0,57	0,90	0,80	0,50	0,00	0,50	0,50
28	0,73	0,79	0,90	0,80	0,00	0,00	0,50	0,50
29	0,78	0,88	0,80	0,90	0,50	0,00	0,50	0,50
30	0,80	0,62	0,90	0,90	0,50	0,80	0,50	0,80
31	0,77	0,83	0,90	0,80	0,00	0,00	0,50	0,50
32	0,83	0,06	0,90	0,80	0,50	0,80	0,50	0,80
33	0,84	0,74	0,90	0,00	1,00	0,00	0,50	0,00
34	0,80	0,69	0,90	0,00	0,50	0,00	0,50	0,50
35	0,78	0,58	0,80	0,80	0,50	0,50	0,50	0,50
36	0,70	0,59	0,80	0,90	0,00	0,00	0,50	0,80
37	0,72	0,71	0,80	0,80	0,00	0,00	0,50	0,50
38	0,82	0,70	0,90	0,80	0,80	0,00	0,50	0,50
39	0,79	0,53	0,90	0,80	0,50	0,00	0,50	0,50
40	0,56	0,66	0,80	0,80	0,50	0,00	0,20	0,50
41	0,74	0,38	0,90	0,80	0,00	0,00	0,50	0,80
42	0,74	0,66	0,90	0,80	0,00	0,80	0,50	0,50
43	0,77	0,71	0,80	0,50	0,50	0,00	0,60	0,20
44	0,86	0,76	0,90	0,90	1,00	0,00	0,50	0,50
45	0,72	0,61	0,90	0,00	0,00	0,00	0,50	0,00
46	0,77	0,71	0,90	0,80	0,50	0,00	0,50	0,50
47	0,84	0,88	0,90	0,90	0,90	0,00	0,50	0,50
48	0,79	0,63	0,90	0,80	0,50	0,00	0,50	0,50
49	0,91	0,80	0,90	0,80	1,00	0,00	0,50	0,50
50	0,83	0,72	0,90	0,80	0,50	1,00	0,50	0,80
51	0,70	0,00	0,90	0,80	0,00	0,00	0,50	0,80
52	0,77	0,00	0,90	0,90	1,00	0,80	0,50	0,50
53	0,77	0,00	0,80	0,80	0,50	0,80	0,20	0,80

Table 6 Evaluation values (ranging from 0 to 1) grouped by criteria and average. value

Repository	Language Grammar		Examples Usage		Readability		Formatting		Structure		Useless Information	
	a3	b3	a4	b4	a5	b5	a6	b6	a7	b7	a8	b8
1	0,80	0,50	1,00	0,00	0,90	0,90	0,80	0,80	0,80	0,80	0,80	0,80
2	0,80	0,80	0,80	0,00	0,90	0,90	0,80	0,50	0,80	0,80	0,80	0,40
3	0,80	0,80	1,00	0,00	0,90	0,90	0,80	0,00	0,90	0,80	0,60	0,00
4	0,80	0,80	1,00	0,80	0,90	0,90	0,80	0,90	0,90	1,00	0,80	0,80
5	0,90	0,90	1,00	1,00	0,90	0,90	0,50	0,00	0,90	1,00	1,00	0,80
6	0,80	0,80	0,50	0,80	0,90	0,90	0,80	0,50	0,80	0,90	0,80	0,80
7	0,80	0,80	1,00	0,50	1,00	0,80	0,60	0,50	0,90	0,90	1,00	0,80
8	0,90	0,80	1,00	1,00	0,90	0,90	0,80	0,90	0,80	0,90	0,80	0,60
9	0,90	0,90	1,00	1,00	0,90	1,00	0,80	0,90	0,90	1,00	0,80	0,20
10	0,80	0,80	1,00	1,00	0,90	0,90	0,80	0,80	0,90	1,00	0,80	0,80
11	0,90	0,80	1,00	1,00	0,90	0,80	0,70	0,75	0,90	0,80	0,60	0,80
12	0,90	0,90	1,00	1,00	0,90	0,90	0,80	0,75	0,90	0,90	0,80	0,80
13	0,80	0,80	1,00	0,50	0,90	0,80	0,80	0,50	1,00	0,80	0,80	0,40
14	0,80	0,80	1,00	0,80	0,90	0,90	0,50	0,80	0,80	0,80	0,60	0,80
15	0,80	0,80	1,00	1,00	0,90	0,90	0,80	0,50	0,90	0,80	0,80	0,80
16	0,80	0,90	1,00	1,00	0,90	0,90	0,80	0,80	0,90	0,90	0,80	0,80
17	0,90	0,90	1,00	0,80	0,90	0,90	0,80	0,90	1,00	0,90	0,40	0,40
18	0,80	0,80	1,00	1,00	0,90	0,90	0,50	0,00	1,00	0,90	1,00	0,00
19	0,90	0,80	0,80	0,00	0,90	1,00	0,90	0,50	0,90	0,90	0,60	0,80
20	0,80	0,90	1,00	0,80	0,90	1,00	0,80	0,90	1,00	1,00	0,80	0,20
21	0,90	0,90	1,00	1,00	0,90	0,90	0,80	0,50	0,90	0,90	0,80	0,80
22	0,80	0,90	1,00	1,00	1,00	0,90	0,80	0,60	1,00	0,90	0,60	0,60
23	0,90	0,90	1,00	1,00	0,90	0,90	0,80	0,50	0,90	0,90	0,80	0,80
24	0,80	0,00	1,00	0,00	0,90	0,00	0,80	0,00	0,90	0,00	0,80	0,00
25	0,80	0,80	1,00	0,60	1,00	0,80	0,80	0,50	0,90	0,90	1,00	0,80
26	0,80	0,80	1,00	0,80	0,90	0,90	0,80	0,50	0,90	0,90	0,80	0,80
27	0,80	0,80	1,00	1,00	1,00	0,90	0,80	0,80	1,00	1,00	0,80	0,80
28	0,80	0,80	1,00	0,00	0,90	0,90	0,80	0,50	0,90	0,80	0,80	0,80
29	0,80	0,90	1,00	1,00	0,90	1,00	0,80	0,90	0,90	0,90	0,80	1,00
30	0,80	0,90	1,00	1,00	0,90	1,00	0,80	0,90	1,00	0,80	0,80	0,80
31	0,80	0,80	1,00	1,00	0,90	0,90	0,80	0,00	1,00	0,80	1,00	0,80
32	0,90	0,90	1,00	1,00	1,00	0,90	0,90	0,50	1,00	1,00	0,80	0,80
33	0,80	0,00	1,00	0,00	0,90	0,00	0,80	0,00	0,90	0,00	0,80	0,00
34	0,90	0,00	1,00	0,00	0,90	0,00	0,80	0,00	0,90	0,00	0,80	0,00
35	0,80	0,80	1,00	1,00	0,90	1,00	0,80	0,50	0,90	0,80	0,80	0,80
36	0,90	0,90	1,00	1,00	0,90	0,90	0,50	0,00	0,90	0,90	0,80	0,80
37	0,80	0,90	1,00	0,00	0,90	0,90	0,80	0,50	0,90	0,80	0,80	0,80
38	0,90	0,80	0,80	0,00	0,90	0,90	0,90	0,80	0,90	0,90	0,80	0,60
39	0,80	0,90	1,00	0,80	0,90	1,00	0,80	0,80	0,90	0,80	0,80	0,80
40	0,50	0,90	0,80	0,80	0,90	1,00	0,50	0,50	0,80	1,00	0,00	0,80
41	0,80	0,80	1,00	0,00	0,90	0,90	0,80	0,20	1,00	0,50	0,80	0,80
42	0,90	0,80	1,00	0,00	0,90	0,90	0,80	0,50	0,90	0,80	0,80	0,80
43	0,80	0,50	0,80	0,00	0,90	0,80	0,80	0,50	0,90	0,50	0,80	0,40
44	0,90	0,90	0,80	0,00	1,00	1,00	0,90	0,80	0,90	1,00	0,80	0,80
45	0,80	0,00	1,00	0,00	0,80	0,00	0,80	0,00	0,90	0,00	0,80	0,00
46	0,80	0,80	1,00	1,00	0,90	0,90	0,80	0,80	0,90	0,80	0,60	0,80
47	0,80	0,90	1,00	1,00	0,90	1,00	0,80	0,80	1,00	0,90	0,80	0,80
48	0,80	0,80	1,00	0,80	0,90	0,90	0,80	0,00	0,90	0,90	0,80	0,80
49	0,90	0,90	1,00	0,80	1,00	0,90	0,90	0,80	1,00	0,90	1,00	0,80
50	0,90	0,90	1,00	1,00	0,90	0,90	0,80	0,80	1,00	0,90	1,00	0,80
51	0,90	0,80	1,00	0,00	0,90	0,90	0,80	0,80	0,90	0,80	0,40	0,80
52	0,80	0,80	1,00	1,00	0,90	0,90	0,50	0,80	0,90	0,90	0,40	0,60
53	0,90	0,80	1,00	0,00	0,90	0,90	0,80	0,80	1,00	0,80	0,80	0,80

Table 7 Evaluation values (ranging from 0 to 1) grouped by criteria and average value.