Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

BACHELOR THESIS

Integrating and Evaluating LLM-Generated Code Documentation in the IDE

submitted by

Hans-Alexander Christian Kruse

Faculty: MIN

Department: Informatics

Group: Applied Software Technology (MAST)

Course: Bachelor of Science Informatics

Student ID: 6799993

Supervisor: Prof. Dr. Walid Maalej

Co-Supervisor: Dr. Steffen Hauf

Mentor: Tim Puhlfürß

Abstract

Context. I developed an extension for VS Code that enables the user to generate documentation for code snippets, using GPT-4, with an engineered *few-shot prompt*.

Objective. The goal was to find out, if *few-shot prompts* do lead to better outputs than human-written prompts, when it comes to code comment generation.

Methodology. I conducted a controlled experiment with 50 participants from both academia and software related industries. In the experiment, the participants were split into a test and a control group. The test group was given the extension I developed and the control group was given a benchmark tool, similar to the ChatGPT web-version. The objective was to rate the generated documentation for two distinct code snippets among six dimensions.

Results. I find that the test tool outperforms the control tool consistently for the dimensions of readability and unnecessary information. Additionally, it outperforms the control tool on the dimensions of helpfulness and usefulness for the more complicated code snippet. I do however find a significant difference in the ratings given, between the students and the non-students in the study.

Zusammenfassung

Kontext. Ich habe eine Erweiterung für VS Code entwickelt, die es dem Benutzer ermöglicht, Dokumentation für Code-Snippets zu generieren, indem GPT-4 mit einem speziell entwickelten *Few-Shot Prompt* verwendet wird.

Zielsetzung. Das Ziel bestand darin herauszufinden, ob *Few-Shot Prompts* zu besseren Ergebnissen bei der Generierung von Code-Kommentaren führen, als Mensch-geschriebene Prompts.

Methodik. Ich führte ein kontrolliertes Experiment mit 50 Teilnehmern aus der Universität und der Softwarebranche durch. Bei dem Experiment wurden die Teilnehmer in eine Testgruppe und eine Kontrollgruppe aufgeteilt. Die Testgruppe erhielt die von mir entwickelte VS Code Erweiterung, während die Kontrollgruppe ein Vergleichstool erhielt, ähnlich der ChatGPT-Webversion. Das Ziel war es, die generierte Dokumentation für zwei unterschiedliche Code-Snippets anhand von sechs Dimensionen zu bewerten.

Ergebnisse. Ich stelle fest, dass das Testwerkzeug in den Dimensionen Lesbarkeit und unnötige Informationen konsistent besser abschneidet als das Kontrollwerkzeug. Zusätzlich übertrifft es das Kontrollwerkzeug in den Dimensionen der wahrgenommenen Hilfe für das Verständnis von Code und der Nützlichkeit, für das komplexere Code-Snippet. Weiterhin finde ich einen signifikanten Unterschied in den vergebenen Bewertungen zwischen den Studierenden und den Nicht-Studierenden in der Studie.

Contents

List of Figures

List of Tables

1 Introduction

In the past year, large language models (LLMs), such as OpenAI's Generative Pre-Trained Transformer (GPT), have gathered significant interest in the software development industry. Among the many proposed use-cases, the generation of code documentation has become a topic that has garnered a lot of attention in the scientific space [1]. This is largely due to the fact that the writing of code documentation has long been a task that is often neglected by developers, for various reasons, including perceived unimportance or lower prioritization [2]. Yet, documenting code has been proven to be among the most important tasks in software engineering, as it enables improved teamwork, as well as easier onboarding for new developers in projects and less accumulated technical debt over time [3–6]. Therefore, automatic code documentation generation with LLMs is a space that offers great potential improvements in how well code is understood and maintained in code bases and projects all over the world.
The goal of this thesis is to find out whether the construction of engineered prompts, based on the latest research in the field, will yield better results than prompts created by developers and students without expert knowledge in the field of prompt engineering. Thus, I aim to answer the following research question:

- **RQ:** Is there a difference in the quality and utility of code documentation generated with human-written prompts vs. a prepared few-shot prompt?

I will draw on the work of Ahmed et al., who have already conducted various studies in the space, using OpenAI's models GPT-3, GPT-3.5, and Codex [7, 8]. I expand that work by examining the newest model in the GPT collection, GPT-4.
The secondary goal is to examine whether LLM-assisted documentation generation tools can offer a significant advantage in the daily workflow of software developers.
To that purpose, I developed an extension for the integrated development environment (IDE) VS Code that enables developers to automatically generate code documentation via GPT-4. I then examined the performance of the tool in a controlled experiment with 50 participants, in which its output was compared to that of a "ChatGPT-like" interface in the IDE, in the form of one of the most popular ChatGPT VS Code extensions. The

generated comments were rated on six dimensions, which have been derived from the latest research in the space of documentation evaluation by Hu et al. [9].
In the experiment, the participants received two distinct code snippets derived from the codebase of a real-world project. The participants were then tasked with generating a comment for each snippet, after which they were asked to rate the comments based on the aforementioned dimensions. Finally, they rated the user experience of the tools, thus giving feedback on perceived improvements they could provide in their workflows.
The experiment shows that the test tool generally performs better among two of the six dimensions and offers a greater user experience than the control tool.
This thesis will first give a concise theoretical overview of LLMs, specifically OpenAI's GPT-4, as well as the newly founded research space of prompt engineering. Then, the creation process of the VS Code Extension will be highlighted, before I give an overview of the methodology of the controlled experiment. Finally, the results of the study will be evaluated and discussed before I highlight some of the drawbacks of the study, as well as further research directions.
This study explicitly only used human evaluation, instead of known metrics such as BLEU, METEOR, etc., to evaluate the generated documentation.
My IDE extension and the experiment results are included in the replication package.

2 Related Work

2.1 Large Language Models

A LLM is a type of artificial neural network designed to understand, generate, and manipulate human language [10]. They are based on the transformer architecture for feed-forward neural networks, first introduced by Vaswani et al. in 2017 [11]. A transformer in this context is a model that employs self-attention mechanisms, which weigh input features by their relevance, thus enabling it to handle longer dependencies in textual data [11].

The training of these models is conducted in two phases: the pre-training phase and the fine-tuning phase. In the pre-training phase, the LLM is trained on large corpora of text input. Depending on the actual model, training data can, therefore, include entire sub-portions of the freely accessible internet. In this phase, the model learns a variety of relevant abilities needed to achieve human-like communication, such as grammar, reasoning abilities, and facts about the world [11].

During the fine-tuning phase, a smaller and more task-specific dataset is used to refine the model for a particular task, such as, for example, code generation in the case of OpenAI's *Codex* model. Due to the amount of data consumed in the pre-training phase, LLMs are very good at generalizing. Though, in practice, especially for more nuanced models, the second mentioned phase is of equal importance [11].

Despite the potential that LLMs hold to achieve significant productivity improvements in various domains, there are still a plethora of challenges to be solved in the space. One of those is the mitigation of bias in the models through their training data [12]. By learning through datasets that consist of unmoderated parts of the internet, LLMs can extract patterns that can be more or less in favor of certain opinions, issues, or even facts, which can be potentially dangerous due to the spread of misinformation or the flawed decision making by LLMs themselves [10].

The second issue is the proneness to over-generalization, which essentially means that an LLM provides information that is factually incorrect due to its overemphasis on broader patterns in the training data, in contrast to more nuanced patterns in test data [13]. This can happen due to the large amounts of training data that can promote converting to an answer that is suitable for usual situations, rather than unusual ones.

2.2 GPT-4

In late 2022, OpenAI's GPT model series gained international attention through the introduction of its Web Tool ChatGPT [14]. It allowed users to directly communicate with the models GPT-3 and later GPT-3.5 by creating an account and entering prompts in the browser [14].
In March 2023, OpenAI introduced the next and, as of the writing of this thesis, most nuanced model in its series, GPT-4. It outperforms GPT-3.5 in tasks of medical examination, as well as legal applications, due to its larger number of parameters and its improved architecture [14]. Additionally, it was trained on an even larger corpus of training data and has knowledge of more recent world events. With this iteration, OpenAI also added the ability for GPT to take images as input and interpret their contents, which enhanced its usability for a wide range of domains.
These enhancements come at the cost of higher computational requirements and higher deployment costs of the model, which is why as of November 2023, the API for GPT-4 and the Web Version are only accessible through a paid plan.
In terms of software engineering-related tasks, GPT-4 has been shown to be proficient at generating code from instructions, executing given code snippets, and offering explanations of their results, among a plethora of other things [10, 15].

2.3 Prompt Engineering

The output of LLMs, such as GPT-4, is controlled via natural language input prompts written by the user. The scientific space of prompt engineering refers to the optimization of the construction of those prompts to achieve more precise outputs [1, 16, 17]. This is particularly important to mitigate the risk of over-generalization, as a carefully constructed prompt can nudge the LLM in the direction of more nuanced parts of its knowledge corpus [16]. Therefore, engineered prompts are proposed to help with achieving better results with LLMs in various subtasks by assisting the models in their alignment with given specific requirements [17].
Recent research has brought up several techniques and patterns in the subspace, such as few-shot prompting, flipped interaction, and persona [7, 17]. For this study, I have focused on applying the few-shot paradigm. In its core, few-shot prompting proposes

the addition of desired example outputs to the human-written prompt [18]. It was first proposed by the authors of the original OpenAI GPT-3 paper by Brown et al. [19] and has since been researched in various domains.
In the context of this study, the most notable prior work has been done by Khan et al. [20] and more recently Ahmed et al. [7, 8]. In their 2022 paper, they outline the potential that few-shot prompting holds in the subtask of code summarization [7]. Most notably, they discuss the advantage of utilizing this method when dealing with "project-specific linguistic phenomena" [7], meaning that each software project holds its own specialized and often unique terminology.
They propose that few-shot prompting a LLM, such as OpenAI's *Codex* in their context, offers a workaround to the data-intensive task of training traditional deep-learning models, such as *CodeBERT*, on a project-specific codebase [7]. In their work, they test this hypothesis by observing and evaluating code summarization output for cross-project as well as same-project datasets.
The results show that in both scenarios, prompting *Codex* with the few-shot paradigm in place outperforms traditional fine-tuned foundation models. At the same time, they find that zero-shot and one-shot prompts do not offer improvements [7].
This is also supported by more recent work from Geng et al. [21], who find that few-shot prompting an LLM achieves results that outperform state-of-the-art approaches in the supervised learning space, while zero-shot and one-shot prompting do not achieve superior results.

2.4 Code Documentation Generation

The space of code documentation generation has gained significant attention in the past decade, with several methods being used and evaluated [22]. Among those are manually crafted templates, information retrieval techniques, as well as more recently deep learning methods [9].
Template-based approaches were among the first developed in the space [23, 24]. They generally focus on the use of Software Word Usage Models (SWUMs) to extract important parts of the method corpus and use it to fill in predefined templates. This approach has been fine-tuned further over the years to take the context of functions into account [24]. While they achieved respectable ratings by human evaluators, their main issue lies within the dependency on the variable names within the method body and the usage of human-written natural language templates [22].
Information retrieval techniques focus on extracting important keywords from code snippets and support documents [25]. They require less generation time compared to

template-based approaches; however, they often only represent partial information [22].
Finally, deep learning-based approaches have gathered more attention in the research space in the last few years [26–30]. Most of them are based on encoder-decoder models that are trained on large datasets and utilize various concepts from Abstract Syntax Trees (ASTs) to plain word representations to gain information from code [22]. More recently, the work has been extended to incorporate transformer-based approaches, as well as hybrid approaches [31].
When it comes to the evaluation of generated documentation, recent research generally distinguishes between automated evaluation metrics and human judgment metrics, with the former being the focus of more recent work in the space [9]. In their 2022 paper, Hu et al. compare the results between automated metrics, such as BLEU, METEOR, ROUGE-L, CIDEr, and SPICE, to results from human evaluation along six distinct dimensions of code comment quality [9]. They split those dimensions into three groups.
The first group consists of language-related metrics, which include naturalness and expressiveness. Naturalness assesses the grammatical correctness and the fluency of the documentation, while expressiveness concerns the readability and the understandability [9].
The second group consists of content-related metrics, which include the adequacy of the content and the conciseness. The former measures the extent to which important information from the source code is reflected in the documentation, while the latter assesses the presence of unnecessary information in the documentation [9].
The final group consists of effectiveness-related metrics, which include the usefulness of the documentation for developers, as well as the understandability improvement the comment provides for the function [9].
In their study, they find that the results of the automated evaluation metrics do not correlate with those of the human evaluation metrics, which is why my study focuses specifically on the human evaluation metrics to complement the existing research, which predominantly focuses on the automated evaluation metrics [29, 32–35].

2.5 Documentation Generation Tools

There currently is a shortage of scientifically examined tools for documentation generation. In their 2022 work, Durelli et al. [36] tested a Visual Studio Code extension that is based on the encoder-decoder pre-trained model *CodeTrans*. The extension offers short summaries of code snippets in Python, Java, and CSharp [36]. To the best of

our knowledge, there is no other IDE extension available that has been developed in a scientific context.
However, there is a growing number of tools available in the public space, with *Github Copilot* being the most popular one [37]. The tool offers real-time auto-complete features when writing comments and also full suggestions for documentation. It does not solely focus on comment generation but instead offers a large number of functionalities to improve workflows for software developers. Additionally, it does not allow the user to adjust the prompt and offer examples for comment styles.
Apart from that, there are many IDE extensions available in the market today that utilize LLMs to help with the process of writing and documenting code. Among those is the tool I used as a benchmark for this study [38].

3 Methodology

3.1 Experimental Design

The Questionnaire for the controlled experiment was split into three parts. For the introductory part, I asked the participants questions about their occupation, their experience with Python, and their experience with the IDE VS Code. The purpose of these questions was to assess the participants' capability of understanding complex code functions, as well as to gain insight into how fast they can navigate VS Code to use the extension optimally. Information such as the person's age was also considered for the survey; however, I opted against that, as its effects on the expressiveness of the study were deemed as non-significant.

The second part of the questionnaire consisted of two sub-parts, each concerning one of two code snippets selected from the European XFEL (EuXFEL) *Karabo* codebase [39]. For each snippet, the participants got approximately 2-3 minutes to review it and try to understand the code to the best of their abilities. I chose this time limit to ensure that participants would not have a chance to understand every last detail of the shown functions, as that would most likely impair their rating for the improvement in understandability.

After the timer ran out, they were asked to rate their understanding of each snippet on a scale of 1 to 5. Additionally, they were given two true/false statements about each snippet to further examine their understanding. For each of these statements, participants also had the option to declare that they had insufficient information to answer the question. Finally, they were asked to write the explanatory part of a function comment for each function by filling out a template I provided them, which can be seen in Figure 3.1. For that template, I adhered to the recommended guidelines for Docstrings in Python 3 [40]. It is also the same template that was used in the few-shots I provided in the prompt.

After this, they were asked to use the provided tool to generate a function comment for the method and study it briefly. Subsequently, they were asked to answer the same two true/false questions as before to assess whether or not the generated comment

```
    '''
        Brief description of the function.

        :param name: Description.
        :type name: type

        :returns: Descritpion.
        :rtype: type
    '''
```

Figure 3.1: Python 3 Comment Template

Dimension	Question
1	Ignoring the content, how would you rate the grammatical correctness of the generated comment?
2	Ignoring the content, how would you rate the readability of the generated documentation?
3	How much relevant information regarding the code snippet do you feel is missing in the generated comment?
4	How much unnecessary information do you feel is present in the generated comment?
5	How useful do you feel the generated comment is for developers?
6	How helpful would you say the generated comment is, to improve the understanding of the code snippet?

Table 3.1: Survey Questions

provided them with enough additional information to improve their answers. The participants were then asked to rate the generated comment based on the six dimensions of code comment quality assessment by Hu et al. [9], which I explained in Section 2.4. I made slight adjustments to the naming of the dimensions to improve the understandability for the participants and avoid longer questions about what they incorporate during the study (Table 3.1).

For the first two dimensions, naturalness and expressiveness, I opted to go for grammatical correctness and readability instead, as those two are mentioned in the paper too and are more expressive for non-native English speakers.

For the second two dimensions, instead of asking the participants to rate the adequacy and the conciseness, I formulated two questions about the amount of missing information and the amount of unnecessary information.

Only the final two dimensions, usefulness and understandability, were not renamed, as they were easy enough to understand.

For each of the dimensions, we provided a five-point answer scale with the lowest point of each scale representing the negative pole, like a *very low readability* and the highest point representing the positive pole, like a *very low amount of unnecessary information*.

The final part of the questionnaire consisted of questions regarding the usability of the provided tool. For that, I opted for the standardized UEQ that measures 26 dimensions of usability in a ranking between 1-7 [41]. Additionally, the participants then had two open text fields to add comments about aspects of the tool they would improve, as well as aspects they found to be particularly useful.

3.2 Technical Setup

3.2.1 Code Snippet Selection

For the code snippets used in the survey, I utilized the connection to the EuXFEL project and selected two distinct methods out of the Python codebase for the *Supervisory Control and Data Acquisition (SCADA)* framework *Karabo* [39]. I opted to primarily select utility functions, as they would be able to be understood without much knowledge of the rest of the codebase. Python was chosen as the programming language for the snippets, as large portions of the codebase are written in Python, and it is a widely used language that all of the participants had at least some knowledge of. Another reason is the fact that recent research has shown Python to be the language that GPT models are most proficient in [1].
After a thorough screening process of all the available functions, I limited the options down to the six functions included in the appendix. From those options, I sorted the functions into the categories easy, medium, and hard together with my mentor, to get a more objective assessment of their perceived difficulty. Based on the categorization, I ended up choosing functions 2, 4 and 6 for the study. However, after conducting a dry-run of the study with select participants, I removed function 6 from the study, as the participants had a hard time understanding it and therefore took a much larger amount of time to complete the survey. That was deemed to be negative, as a longer expected time for the survey completion could deter people from participating. I therefore concentrated on just functions 2 and 4 (Figure 3.2).

```python
2. def string_from_vector_bool(data):
        return ",".join(str(int(i)) for i in data)
```

```python
4. def _parse_date(date):
        if date is None:
            date = Timestamp()
        if isinstance(date, Timestamp):
            date = date.toLocal()
        d = dateutil.parser.parse(date)
        if d.tzinfo is None:
            d = d.replace(tzinfo=dateutil.tz.tzlocal())
        return d.astimezone(dateutil.tz.tzutc())
        .replace(tzinfo=None).isoformat()
```

Figure 3.2: Karabo Python Functions

The purpose of function 2 is the conversion from a boolean vector into a comma-separated string. To that purpose, a for loop runs through the boolean vector and firstly converts each of the true/false elements into the corresponding integer value of 1/0 and then turns the integers into strings. The extracted values are then joined together, with a comma being chosen as the separating character.
Function 4 is a date parsing function, which has the purpose of converting a given date into a Python datetime object in *ISO* format. To that purpose, the function first checks if the argument given into it is `None`. If that is the case, it assigns it an instance of the `Timestamp` class. Herein lies the first difficulty of function 4 in contrast to function 2, as I provide no information about what the `Timestamp` class includes. Therefore, the participants, as well as GPT-4, have to make assumptions based on the name and the context of the usage of it. Instances of the class hold indeed information about the current time in attoseconds since the epoch [39]. After the first if statement, the function checks if the given date is an instance of the `Timestamp` class. If that is true, it transforms the date to local time using the `.toLocal()` function. Next, it parses the date into a Python datetime object and saves it in the variable `d`, before checking whether or not there is already some timezone info available in the created object. If that is not the case, the function adds local timezone information to it. Finally, the datetime object `d` is converted to *UTC* timezone using `astimezone(dateutil.tz.tzutc())` and then the timezone information is stripped off using `.replace(tzinfo=None)`. The resulting datetime object is then converted to an *ISO* format string using `.isoformat()` and returned [39].

3.2.2 Model Selection

Over the course of the past year a plethora of LLMs have been introduced to the public. Therefore a key part of this thesis is the selection of a model. For this study I decided on OpenAI's GPT-4 as the examined model. This decision was made based on the popularity of the model, as well as the prevalence on recent research in the comment generation space using GPT-3, 3.5 and Codex [13]. Even though the research on GPT-4s prevalence is still relatively new, it was deemed as interesting to see, how the model performs in comparison, especially to GPT-3.5 [10]. Additionally, when comparing it to other LLMs that were released in the timeframe of the writing of this thesis, GPT-4 proved generally superior, to other candidates such as *PaLM* and *LLaMA* [14].
OpenAI also hosts the GPT models and provides an easy-to-use application programing interface (API) for my downstream application.
Finally, considering GPT's popularity and rapid evolution, I anticipated that future model

```
"""
For the following prompt take into account these 3 input/output pairs,
for functions and appropriate comments for them: {FEW_SHOTS}

Generate a comment for the following function:{FUNCTION_CONTENT}.

Adhere to the appropriate comment syntax for multiline comments.
Fill out this given template: {TEMPLATE}
"""
```

Figure 3.3: Prompt constructed for the experiment

```
    def elapsed_tid(cls, reference, new):
        """
        Calculate the elapsed trainId between reference and newest
   timestamp

        :param reference: the reference timestamp
        :param new: the new timestamp

        :type reference: Timestamp
        :type new: Timestamp

        :returns: elapsed trainId's between reference and new timestamp
        """
        time_difference = new.toTimestamp() - reference.toTimestamp()
        return np.int64(time_difference * 1.0e6 // cls._period)
```

Figure 3.4: Few-Shot 1

versions would further improve in code documentation generation and, consequently, benefit documentation-oriented tools.
In fact, subsequent to this thesis, OpenAI released a GPT-4 version with an expanded context window size, potentially allowing the generation of more context-sensitive code comments.

3.2.3 Prompt Construction

I constructed a structured prompt to communicate with GPT-4. The prompt was written following the prompt pattern catalogues in recently published research, as well as the recommended guidelines published by OpenAI [17, 35, 42]. One of the most prominent forms of prompting is few-shot prompting, where the user gives the LLM a couple of example input/output pairs to fine-tune it on expected answers [17]. Due to the limit of GPT-4's API at the time of the study, I opted for 3-shots, meaning that I handed GPT-4 three distinct functions, along with comments as examples, alongside

```
    def get_array_data(data, path=None, squeeze=True):
        """
        Method to extract an ``ndarray`` from a raw Hash

        :param data: A hash containing the data hash
        :param path: The path of the NDArray. If `None` (default) the
    input Hash is taken.
        :param squeeze: If the array should be squeezed if the latest
    dimension is 1. Default is `True`.

        :returns: A numpy array containing the extracted data
        """
        text = "Expected a Hash, but got type %s instead!" % type(data)
        assert isinstance(data, Hash), text
        array = _build_ndarray(data, path=path, squeeze=squeeze)
        return array
```

Figure 3.5: Few-Shot 2

the rest of the prompt.

The selection of those functions was based on criteria for code comment quality derived from the recent study by Rani et al. [43]. Among metrics such as conciseness, completeness, and usefulness, they find that oftentimes code comments are written in differing styles, depending on preferences of the project team [43]. Therefore, I opted to choose three code and comment pairs from the same project codebase as the two snippets that the participants were supposed to evaluate [39]. During the screening process of possible pairs, I adhered to the metrics derived from the literature. After careful consideration, I decided on the three code and comment pairs shown in Figures 3.4, 3.5, and 3.6. They each follow the same template that is used throughout the project and offer examples for GPT-4 as to the desired length and format of the output.

The rest of the prompt was constructed as straightforward as possible. I ask GPT-4 to write a comment for the given function and pass the snippet along. I then urge it to adhere to the multiline syntax for Python comments and Docstrings, and I provide it a template to make sure that the returned comment is in a desired format (Figure 3.3). Through a secondary functionality of the tool, the user would later be able to change the prompt, but most importantly the template for the comment to adhere to different kinds of preferred comment formats.

3.2.4 Tool Implementation

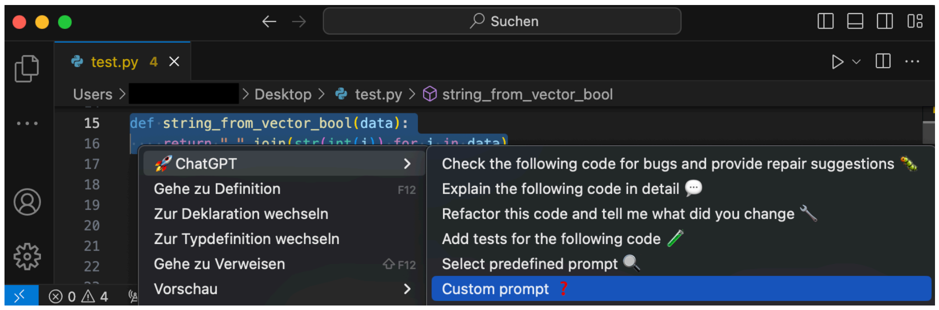
To examine the documentation generation capabilities of LLMs such as GPT-4, I created an extension for the IDE VS Code. The IDE was chosen based on its popularity and its

```python
    async def getSchema(device, onlyCurrentState=False):
        """
        Get a schema from a target device

        :param device: deviceId or proxy
        :param onlyCurrentState: Boolean for the state dependent schema. The default is `False`.

        :returns: Full Schema object
        """
        if isinstance(device, ProxyBase):
            if not onlyCurrentState:
              return Schema(name=device.classId, hash=device._schema_hash)
            else:
                device = device._deviceId

        schema, _ = await get_instance().call(device, "slotGetSchema",
                                          onlyCurrentState)
        return schema
```

Figure 3.6: Few Shot 3

prevalence in many software engineering-related fields. Furthermore, VS Code offers support for a large portion of popular programming languages, and most importantly, for the context of this thesis, for Python3. Additionally, VS Code already offers a number of user-written extensions that incorporate LLMs such as GPT-4, which allowed us to freely choose a tool whose output I could use as the benchmark.
The main criterion for the selection of the benchmark tool was that it should be relatively similar to the ChatGPT Web interface, to enable the users to write their own prompts and to compare the potential improvement to the usability by automatically adding the comments above the function in comparison to the process of copying and pasting them from GPT's output window. After extensive testing of available extensions, I decided on comparing my tool to the *ChatGPT* extension, written by *Zhang Renyang* [38]. The extension satisfies the requirement of offering a ChatGPT-like chat window that opens beside the code editor, and it is also among the most popular GPT extensions available in the store. Additionally, it allows the use of GPT-4 through providing an API key, which in turn allowed me to compare the outputs from participant-written prompts to the outputs from the prompt I wrote. The general workflow for writing a comment with the extension can be seen in Figures 3.7-3.9.

For the creation of my extension, the documentation for the generation of VS Code extensions was followed thoroughly [44]. To that purpose, the necessary libraries *yeoman* and *generator-code* were installed locally using the Node Package Manager (npm).

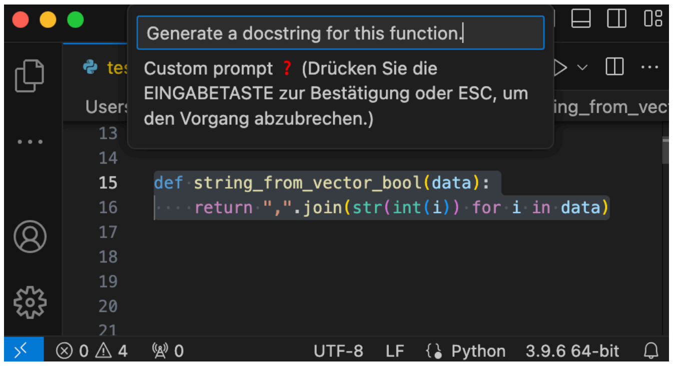


Users > [redacted] > Desktop > test.py > string_from_vector_bool

```
def string_from_vector_bool(data):
```

Suchen
test.py 4
ChatGPT
Gehe zu Definition F12
Zur Deklaration wechseln
Zur Typdefinition wechseln
Gehe zu Verweisen ⇧F12
Vorschau
Check the following code for bugs and provide repair suggestions
Explain the following code in detail
Refactor this code and tell me what did you change
Add tests for the following code
Select predefined prompt
Custom prompt

Figure 3.7: Selection of the command for the control tool

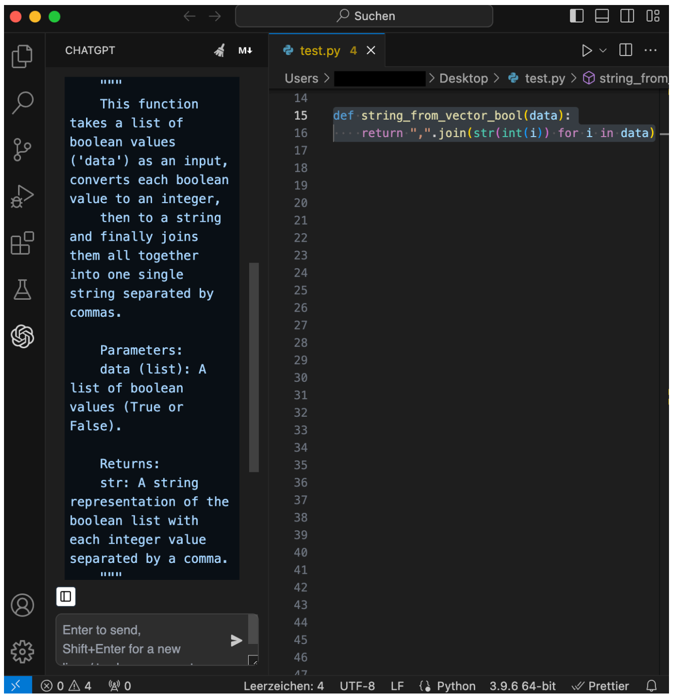


Generate a docstring for this function.

Custom prompt ❓ (Drücken Sie die EINGABETASTE zur Bestätigung oder ESC, um den Vorgang abzubrechen.)

```
def string_from_vector_bool(data):
    return ",".join(str(int(i)) for i in data)
```

Figure 3.8: Writing of the prompt for the control tool

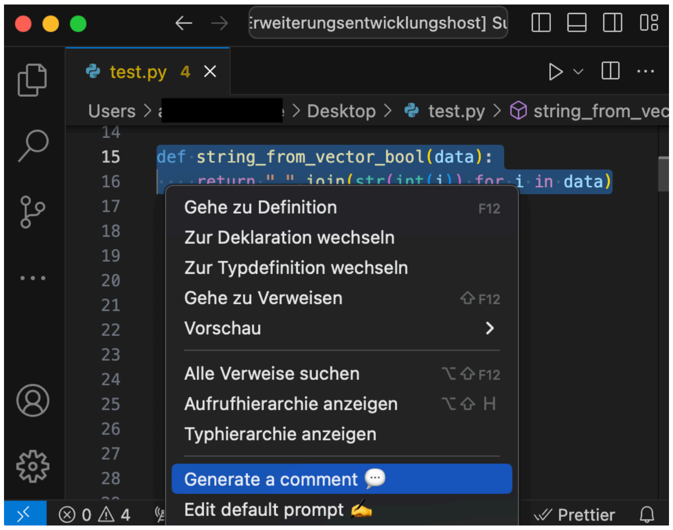


```
    """
    This function takes a list of boolean values ('data') as an input, converts each boolean value to an integer,
    then to a string and finally joins them all together into one single string separated by commas.

    Parameters:
    data (list): A list of boolean values (True or False).

    Returns:
    str: A string representation of the boolean list with each integer value separated by a comma.
    """
```

Users > [redacted] > Desktop > test.py > string_from

```
def string_from_vector_bool(data):
    return ",".join(str(int(i)) for i in data)
```

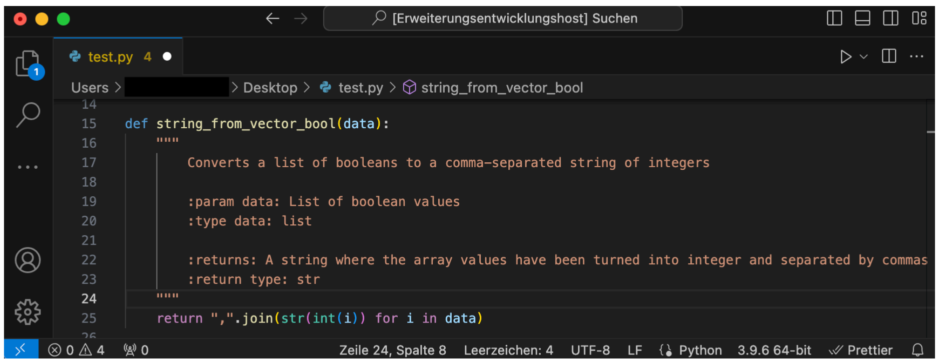
Figure 3.9: Generating the output for the control tool



```
def string_from_vector_bool(data):
    return "".join(str(int(i)) for i in data)
```

Figure 3.10: Selection of the command for the test tool

As a programming language for the extension, I used *TypeScript* rather than *JavaScript*, as the type safety that it offers was deemed to be important.
The aforementioned libraries offer boilerplate code for the project, which will not be further discussed in this thesis, as its workings are covered extensively on the Visual Studio Code Website [44].
For the selection of the function or class the user would like to generate a comment for, I opted for a simple selection of the needed space, as it offered the most precision in contrast to taking the entire file as context for the documentation. I then added the ability to generate the comment either by entering the command "Generate Comment" and then choosing the appropriate comment type or by simply using the context menu and selecting the option there. The idea behind that was to enable inexperienced VS Code users to use the extension through a known path, while giving experienced users the ability to use the command palette, thus integrating the extension's ability into their usual workflow.
To make the API call to the OpenAI API, I used a simple *axios* request, as the use of dedicated libraries, such as *Langchain*, was not possible in a VS Code Extension environment at the time of the development. Additionally, the usage did not offer any further functionality in this use-case. GPT-4 then replies with a dedicated comment, which is automatically inserted below the head of the selected function. Figures 3.10 and 3.11 show the general workflow.


[Erweiterungsentwicklungshost] Suchen
test.py 4
Users > [redacted] > Desktop > test.py > string_from_vector_bool

```python
def string_from_vector_bool(data):
    """
        Converts a list of booleans to a comma-separated string of integers

        :param data: List of boolean values
        :type data: list

        :returns: A string where the array values have been turned into integer and separated by commas
        :return type: str
    """
    return ",".join(str(int(i)) for i in data)
```

0 4 0 Zeile 24, Spalte 8 Leerzeichen: 4 UTF-8 LF Python 3.9.6 64-bit Prettier

Figure 3.11: Generating the output for the test tool

3.3 Experiment Execution

The survey was conducted in a two-month timeframe, between October and November 2023. I used GPT-4 version *gpt-4-0613* in both tools, with VS Code Version *1.85* and *ChatGPT* Extension version *1.6.62*. 50 people participated in the study. 30 of those were students, studying subjects from computer science, to business informatics and software systems engineering. The other 20 participants, were people working in software development and related fields, such as finance or software research. All of them had at least some experience with software development, in various programming languages, which was the only requirement for participating in the experiment. Table 3.3 shows the responses of the participants for the questions regarding their prior experience with Python and VS Code.
Most of the interviews were conducted via the online video platform Zoom, while 17 of them were conducted on-site at EuXFEL's complex in Schenefeld, Germany.
In addition to the questionnaire, the interviewer urged the participants to utilize a *Thinking Aloud* approach [45]. This was deemed as especially important for the part of the survey, in which the participants had to rate the comments because, as I observed in the dry-run of the study, I was able to gain valuable insights into what the participants thought was especially important in the generated comments.
Another important aspect for the interviewer during the study was the prompts written by the people in the control group. They were noted during the interviews for further comparison in the results section of this thesis and can be found in the replication

Table 3.2: Participants' occupations in Test / Control groups

	Students	**Software Engineers**	**Data Scientists**	**Others**
T / C	15 / 15	6 / 6	2 / 2	2 / 2

Table 3.3: Participants perceived Python experience and VS Code experience on a scale from 1-5

Experience	**1**	**2**	**3**	**4**	**5**
Python	5	13	10	14	8
VS Code	5	10	12	20	3

package.
On average, the participants in the test group completed the experiment in 19 minutes and 57.6 seconds, while the participants in the control group needed 23 minutes and 52.8 seconds. This slightly higher time can be explained by the additional work of writing prompts that the control group had.

3.4 Data Analysis

The evaluation of the collected data included quantitative as well as qualitative aspects.
For the quantitative part, I aimed to utilize either a parametric *Welch's t-test* or a non-parametric *Mann-Whitney U (MWU)* test to test for significant differences between the ratings for the six dimensions among the test and control groups [46, 47]. I complemented the results with average means for all the dimensions among the groups. Additionally, I split the participants into groups twice. Firstly, for students/non-students, to test if their general experience writing code had an influence on the ratings. And secondly, for low/high experience with the programming language Python, to test if this had an influence on the ratings. I further complemented the quantitative insights gained from the tests with qualitative insights gained during the *Thinking Aloud* phase of the experiment. To measure the improvement to function understanding, I tested how many people in both groups answered the true/false statements in the survey correctly and if the tool influenced their decisions. All calculations utilized functions from the *SciPy Statistical Functions* module [48].
For the prompts written by the control group participants, I compared them to each other as well as to the prompt used for the test tool and highlighted important differences and some similarities.
Finally, I utilized the Excel tool provided by the UEQ Team [41] to analyze the UEQ data [49]. The tool offered descriptive (mean values and standard deviations) and inferential statistics (t-test) to compare user experience ratings for the two IDE extensions.

Additionally, I manually analyzed participants' free-text answers to the questions “What did you like most about the tool?” and “What would you change about the tool?” at the end of the questionnaire.
My mentor and I independently conducted *Open Coding* for all answers and compared their codes to achieve consensus in reporting [50].

4 Results

4.1 Prompt Comparison

When it comes to the prompts entered by the participants in the control group, there were significant differences, to the prompt used in the test tool. Most of the prompts followed one of three wordings that can be seen in Table 4.1 (a detailed version is in the replication package). Among the 50 prompts entered by the participants throughout the study, there were only three that were too specialized to be assigned to one of the most frequent groups, and just one prompt was longer than one sentence. No participant thought of enhancing their prompt with example input/output pairs or a comment template. This might, in part, be due to the small input field the control tool provided, which nudged the users to write a shorter prompt, and, in part, due to the missing experience when it comes to writing prompts. Another factor could be the slight language barrier. Most of the participants were not native English speakers, but they were asked to prompt in English to make the output more comparable to the test tool.

Generally, the prompts written were rather short, with 60 percent of them being 6-8 words long.

Among the prompts listed, I only included those that actually led to comments as output. About half of the participants had to re-enter a different prompt to get an output resembling a comment, which led to an impaired user experience, as I will discuss later.

Table 4.1: Frequency of successful prompts in the control group, subdivided by occupations

Prompt pattern	Students	Professionals
Write a function comment	**13**	8
Write a comment	12	2
Prepare a Docstring	3	**9**
Other	2	1

Among the prompts that did not lead to a comment as the output were wordings such as "explain the function" and "describe what this function does," which are expected to give suboptimal results. Surprisingly, though, some of the prompts from Table 4.1 also led to an undesired output in a few cases.
I found that there are a couple of keywords that proved to be especially important in regards to even getting a comment as output. Those were, as would be expected, "function comment" and "Docstring." In all cases where participants asked specifically for a "function comment" or a "Docstring," the LLM provided them with, at the very least, the desired output format.

4.2 Output Comparison

Next, I compare the different ratings among the six dimensions for the quality of the generated comments.
First, I conducted a *Shapiro-Wilk test* to test if the data for each scale in each group follows a normal distribution [51]. I found that none of our scales satisfy that requirement, most likely due to the small sample size. Next, I conducted a *Levene's test* to test the homogeneity between the variances in the groups. I find that only for snippet 2, dimensions 1-4, the variances are not equal.
As a result of those insights, I decided against conducting a *Welch's t-test* and opted for the *MWU test* instead, as the test is recommended for scenarios in which there is no notable normal distribution among the dimensions [47].
Among the six dimensions derived from the paper by Hu et al. [9], I obtained the following results on a 0.05 significance level for the first and the second code snippet, respectively.
For snippet 1, I found that dimensions 2 and 4 had a significant difference in rating among the two groups. Dimension 2, the readability of the generated documentation, had an average rating of 4.72 in the test group and 4.12 in the control group. For dimension 4, the test group scored the amount of unnecessary information on the lower side, with 4.32 on average, and the control group scored the amount to be in the middle of the scale with 3.28. This coincides with the observations made during the study, where the biggest difference between the documentation generated by the test tool and the documentation generated by the control tool was the length. This, of course, led to subpar readability when it came to the control tool output, as the comment did not follow a particular template. Additionally, the length of the output, of course, led to a lot of unnecessary content, especially for the smaller of the two code snippets, which explains the results.

Table 4.2: Mean ratings for six dimensions in test and control groups for snippet 1

Dimension	Grammatical Correctness	Readability	Missing Information	Unnecessary Information	Usefulness	Helpfulness
Test Group Mean (Std)	4.88 (0.33)	4.72 (0.54)	4.36 (1.11)	4.32 (1.25)	4.44 (1.04)	4.32 (1.07)
Control Group Mean (Std)	4.88 (0.33)	4.12 (0.88)	4.64 (0.76)	3.28 (1.21)	4.12 (0.83)	4.52 (0.59)
MWU-test p-value	1.0	**0.005**	0.348	**0.0009**	0.084	0.912

Table 4.3: Mean ratings for six dimensions in test and control groups for snippet 2

Dimension	Grammatical Correctness	Readability	Missing Information	Unnecessary Information	Usefulness	Helpfulness
Test Group Mean (Std)	4.76 (0.66)	4.68 (0.48)	4.16 (0.85)	4.68 (0.85)	4.6 (0.76)	4.6 (0.82)
Control Group Mean (Std)	4.48 (0.92)	3.52 (1.39)	3.92 (1.29)	2.84 (1.60)	3.52 (1.39)	4.00 (1.08)
MWU-test p-value	0.119	**0.0004**	0.781	**0.00002**	**0.002**	**0.016**

For snippet 2, I found that four of the six dimensions had a significant difference in rating, with the test group consistently scoring higher than the control group. For dimension 2, the readability of the generated documentation, the test group had an average rating of 4.68, while the control group rated it with 3.52. This further drives the point made previously about the missing template, as well as the extensive length. Dimension 4, the amount of unnecessary information, was rated with 4.68 by the test group and 2.84 by the control group, once again coinciding with the observations made for snippet 1. The fifth dimension, the usefulness, was rated with 4.6 by the test group, while the control group rated it with 3.52. This can most likely also be attributed to the excessive length of the control tool output. During the *Thinking Aloud* part of the survey, a large portion of the participants voiced the concern that larger comments could hinder the workflow for developers, as they take up too much time to read for too little additional necessary information. Finally, the sixth dimension, the helpfulness to improve the understanding of the snippet, was rated with 4.6 by the test group and 4.0 by the control group. While this could also be attributed to the bloatedness of the control tool output, according to the participants, the provided explanations were most likely simply too complicated to understand.

Next, I split the participants from both groups into subgroups of students and people working in software engineering-related fields and performed a MWU test for the subgroups to test the results for experienced developers and students separately. I obtained the following results on a 0.05 significance level for the first and the second code

Table 4.4: Mean ratings for six dimensions in test and control groups for the students for snippet 1

Dimension	Grammatical Correctness	Readability	Missing Information	Unnecessary Information	Usefulness	Helpfulness
Test Group Mean (Std)	4.93 (0.26)	4.66 (0.62)	4.6 (0.83)	4.86 (0.35)	4.6 (0.63)	4.6 (0.63)
Control Group Mean (Std)	4.86 (0.35)	4.0 (1.0)	4.6 (0.63)	3.33 (1.35)	3.93 (0.88)	4.46 (0.64)
MWU-test p-value	0.577	**0.032**	0.757	**0.0002**	**0.026**	0.517

Table 4.5: Mean ratings for six dimensions in test and control groups for the students for snippet 2

Dimension	Grammatical Correctness	Readability	Missing Information	Unnecessary Information	Usefulness	Helpfulness
Test Group Mean (Std)	4.73 (0.80)	4.6 (0.51)	4.06 (0.96)	4.86 (0.35)	4.73 (0.46)	4.6 (0.74)
Control Group Mean (Std)	4.4 (1.06)	3.2 (1.47)	3.86 (1.30)	2.66 (1.68)	3.13 (1.41)	3.66 (1.23)
MWU-test p-value	0.133	**0.002**	0.843	**0.0002**	**0.0008**	**0.022**

snippet, respectively, in the subgroups of students.

In contrast to the entire participant test, for snippet 1, the students in the test and control groups scored dimensions 2, 4, and 5 significantly differently from each other. Test group students rated the readability of the generated comment as 4.66 on average, while the control group rated it as 4.00. For dimension 4, the students in the test group scored it as 4.86, in contrast to the control group students with 3.33. For dimension 5, the test group students had an average rating of 4.6, while the control group students had an average rating of 3.93. Thus, we can already see a slight difference in the general results when it comes to the ratings of the students for snippet 1. This could be attributed to the lower experience level of students when it comes to writing documentation.

The rating of snippet 2 reflects the general results, with dimensions 2, 4, 5, and 6 being significantly better rated in the test group. Dimension 2 scores an average of 4.6 for the test group, with the control group having an average rating of 3.2 when it comes to the readability of the generated documentation. For dimension 4, the students in the test group rated it with 4.86 on average, while the control group rated it with 2.66. This is in line with the results for the general study, where the dimension of unnecessary content in the comment is the biggest difference between the outputs generated in the test and control group. The average rating for dimension 5, the usefulness, is 4.73 in

Table 4.6: Mean ratings for six dimensions in test and control groups for the non-students for snippet 1

Dimension	Grammatical Correctness	Readability	Missing Information	Unnecessary Information	Usefulness	Helpfulness
Test Group Mean (Std)	4.8 (0.42)	4.8 (0.42)	4.0 (1.41)	3.5 (1.65)	4.1 (1.45)	3.9 (1.45)
Control Group Mean (Std)	4.9 (0.32)	4.3 (0.67)	4.7 (0.95)	3.2 (1.03)	4.4 (0.70)	4.6 (0.52)
MWU-test p-value	0.583	0.072	0.084	0.511	0.966	0.377

Table 4.7: Mean ratings for six dimensions in test and control groups for the non-students for snippet 2

Dimension	Grammatical Correctness	Readability	Missing Information	Unnecessary Information	Usefulness	Helpfulness
Test Group Mean (Std)	4.8 (0.42)	4.8 (0.42)	4.3 (0.67)	4.4 (1.26)	4.4 (1.07)	4.6 (0.97)
Control Group Mean (Std)	4.6 (0.70)	4.0 (1.15)	4.0 (1.33)	3.1 (1.52)	4.1 (1.20)	4.5 (0.53)
MWU-test p-value	0.582	0.062	0.903	0.067	0.465	0.278

the test group, while it remains at 3.13 for the control group. Finally, dimension 6, the helpfulness to improve the understanding, is rated with 4.6 on average by the students in the test group, while it achieves an average rating of 3.66 by the students in the control group. In general, the average ratings by the students in the control group were consistently lower than the average ratings by the whole control group, indicating that the subgroup of people working in software development-related fields scored the dimensions higher for snippet 2. In line with the general results, the students had the most issues with problems resulting from the length and the complexity of the generated documentation.
For the non-student subgroup, I obtained the following results.
For the MWU test in the subgroup of people working in software development-related fields, I find not a single dimension to yield significantly better ratings in the control group or the test group, for neither snippet 1 nor snippet 2, which presents a contrast to the general group results. It is important to note, however, that for snippet 2, dimension 4 does have a rather large difference in the means for the ratings, with the test group scoring it 4.4, while the control group scores it 3.1. This is not supported by the p-value of 0.067, though, so the result is not generalizable. This might be in part due to the smaller sample size of the non-student subgroup, which consists only of 20 people, in contrast to the 30 people in the student subgroup. Another interesting ob-

Table 4.8: Mean ratings for six dimensions in test and control groups for the lower experience group for snippet 1

Dimension	**Grammatical Correctness**	**Readability**	**Missing Information**	**Unnecessary Information**	**Usefulness**	**Helpfulness**
Test Group Mean (Std)	4.875 (0.35)	4.625 (0.74)	4.75 (0.46)	4.875 (0.35)	4.375 (0.74)	4.875 (0.35)
Control Group Mean (Std)	4.8 (0.42)	4.1 (0.88)	4.8 (0.63)	3.8 (1.23)	4.3 (0.82)	4.6 (0.70)
MWU-test p-value	0.731	0.128	0.538	**0.018**	0.923	0.391

Table 4.9: Mean ratings for six dimensions in test and control groups for the lower experience group for snippet 2

Dimension	**Grammatical Correctness**	**Readability**	**Missing Information**	**Unnecessary Information**	**Usefulness**	**Helpfulness**
Test Group Mean (Std)	4.5 (1.07)	4.75 (0.46)	4.5 (0.53)	4.875 (0.35)	4.875 (0.35)	4.75 (0.71)
Control Group Mean (Std)	4.4 (1.26)	2.8 (1.48)	3.9 (1.20)	2.5 (1.78)	2.9 (1.37)	3.7 (1.16)
MWU-test p-value	0.866	**0.003**	0.361	**0.005**	**0.004**	**0.031**

servation is the higher scoring for snippet 1, dimension 3 by the control group. This is in contrast to all the other observations made. A possible explanation for this is that the test tool output did not always mention how the operations utilized in the snippet worked in detail. The control tool was much more precise and longer in its explanations, which could have led to some information missing in the test tool.
Finally, I split the participants into two other subgroups based on their given answers to the question about their prior experience with using Python to find out whether or not experience with the programming language had any influence on the perceived ratings for the generated documentation. I opted for a split that puts the people that rated their experience with 1 or 2 into one group and the people that rated their experience with 3, 4, or 5 into another. In the following, I will refer to the first group as the lower experience group and the second group as the higher experience group. I once again performed a MWU test and obtained the following results on a 0.05 significance level for the lower experience group.

For snippet 1, I can see a similar result to that of the student subgroup, as well as the general participant group, with dimension 4 showing the only significantly different rating. The lower experienced participants in the test group rated the amount of unnecessary content in the generated comment on the lower side with 4.875, while the low-experienced participants in the control group rated it with 3.8 on average.

Table 4.10: Ratings for six dimensions in test and control groups for the higher experience group for snippet 1

Dimension	Grammatical Correctness	Readability	Missing Information	Unnecessary Information	Usefulness	Helpfulness
Test Group Mean (Std)	4.88 (0.33)	4.76 (0.44)	4.18 (1.29)	4.06 (1.43)	4.41 (1.18)	4.05 (1.20)
Control Group Mean (Std)	4.93 (0.26)	4.13 (0.92)	4.53 (0.83)	2.93 (1.10)	4.00 (0.85)	4.46 (0.52)
MWU-test p-value	0.654	**0.024**	0.539	**0.011**	**0.041**	0.534

Table 4.11: Ratings for six dimensions in test and control groups for the higher experience group for snippet 2

Dimension	Grammatical Correctness	Readability	Missing Information	Unnecessary Information	Usefulness	Helpfulness
Test Group Mean (Std)	4.88 (0.33)	4.65 (0.49)	4.00 (0.94)	4.59 (1.00)	4.47 (0.87)	4.53 (0.87)
Control Group Mean (Std)	4.53 (0.64)	4.00 (1.13)	3.93 (1.39)	3.06 (1.49)	3.93 (1.28)	4.20 (1.01)
MWU-test p-value	0.067	0.054	0.750	**0.003**	0.169	0.228

The findings for snippet 2 also fall in line with those of the general study, in which dimensions 2, 4, 5, and 6 show significant differences in rating, with dimension 2 being rated with a 4.75 on average by the lower experienced people in the test group and a 2.8 by the people in the control group. Dimension 4 and 5 each got an average rating of 4.875 in the test group, versus the 2.5 and 2.9 ratings, respectively, by the control group, while dimension 6 was rated with a 4.75 on average in the test group and a 3.7 on average in the control group. Especially the fifth dimension, the perceived usefulness for developers, shows a significantly lower average rating in this subgroup than in the general study.

For the high experience subgroup, for snippet 1, I find that, in line with the general results, dimensions 2 and 4 yield significantly better results in the test group than in the control group. For dimension 2, the test group had an average rating of 4.76, while the control group had an average rating of 4.13. Dimension 4 had an even clearer difference in means, with the test group scoring the amount of unnecessary information in the comment with 4.06, while the control group scored it with 2.93. In contrast to the general results, dimension 5 also proves to be significantly different in both groups, with the test group scoring it 4.41 and the control group scoring it 4.00 on average. Another interesting observation can be made in dimension 6, where the control group had a higher average rating than the test group. This could be attributed to the fact

that most participants who had some experience working with Python already had a very good understanding of snippet 1 without the tool. Therefore, to rate the improvement to their understanding, they required much more detailed information about the snippet, which only the control tool provided.
For snippet 2, I find a significantly different outcome than for the general results, with only dimension 4 showing a significant difference in rating, with the test group scoring an average of 4.59 versus the average rating of 3.06 in the control group. In contrast to the ratings for snippet 1, the higher experienced participants scored the improvement to their understanding higher for the test tool. This is most likely the case because even for highly experienced developers, the function proved to be quite difficult to understand without proper context. Therefore, they reverted to their standards for code documentation and did not gain any additional insights through the more detailed explanations of the control tool.

4.3 Function Understanding

First, I check the function understanding before the participants were given a tool to generate a comment (Table 4.12). In general, most of the participants had a much better understanding of snippet 1 in comparison to snippet 2, which is in line with the expectations, as the first snippet is less complex.

Through the true/false questions in the questionnaire, I was able to get a further overview of the participants' understanding of the given code snippets. To that purpose, I calculated the number of participants who answered the questions correctly before and after using each of the tools and obtained the following results (Table 4.13). Before using the tool, most of the participants in both groups answered the questions for snippet 1 correctly. This was expected, as snippet 1 was generally relatively short and easy to understand.
I observe a relatively counterintuitive result for the rate of correctly answered questions after using both tools, as almost all the percentages go down. Taking the observations into account, a relatively large number of participants actually got confused by the out-

Rating	**TS1**	**CS1**	**TS2**	**CS2**
Mean (Std)	3.84 (1.11)	2.92 (0.81)	3.6 (0.96)	2.4 (0.96)

Table 4.12: Comprehension of code snippets 1 and 2 (S1/2) by test and control groups (T/C); mean values, standard deviations before using any tool

Group	S1 Q1 B	S1 Q2 B	S1 Q1 A	S1 Q2 A	S2 Q1 B	S2 Q2 B	S2 Q1 A	S2 Q2 A
Test Group	92.0%	96.0%	96.0%	80.0%	56.0%	52.0%	72.0%	84.0%
Control Group	76.0%	92.0%	88.0%	84.0%	40.0%	60.0%	88.0%	84.0%

Table 4.13: Percentage of correct answers for the questions (Q1, Q2) for snippet 1 (S1) and 2 (S2), before (B) and after (A) using a tool

put of both tools. This could be explained by the fact that the question asked if the input to the function is an *Iterable*. Both outputs generally used less general terms for the input, which most likely confused the participants with less experience in Python. For snippet 2, the percentages are much lower, which was expected, as the function was more complicated and required a degree of additional knowledge to be understood entirely. I can see a rather large general improvement for both groups in both questions. The control tool actually achieved the largest jump for question 1. This is most likely a positive factor of the longer and more detailed comments the control tool generated. While they may not be ideal for developers working on the code, they certainly outperform the test tool for conveying more details about the snippet. This seems counterintuitive to the slightly better ratings the test tool received in the category of understandability, which leads to the conclusion that understandability in itself can be split into two different forms: the general understanding of what the function does and the understanding of smaller details, such as those asked for in the true/false questions.

4.4 User Experience Evaluation

4.4.1 UEQ

Finally, I evaluated the user experience through the standardized UEQ [41]. Figure 4.1 shows that the test tool proves to be superior on five of the six measured dimensions

Table 4.14: Comparison of user experiences: Ratings of the six UEQ categories (mean values and standard deviations) from the test and control groups (T/C), including the t test's statistical significance indicator (p-value)

Dimension	Attractiveness	Perspicuity	Efficiency	Dependability	Stimulation	Novelty
Test Group Mean (Std)	1.99 (0.63)	1.87 (0.47)	1.98 (0.67)	1.84 (0.79)	1.62 (0.83)	0.95 (0.77)
Control Group Mean (Std)	1.01 (1.14)	1.25 (0.68)	0.70 (1.19)	1.05 (1.05)	0.82 (1.13)	1.10 (1.15)
t-test p-value	**0.0006**	**0.0005**	**0.0000**	**0.0043**	**0.0065**	0.5909

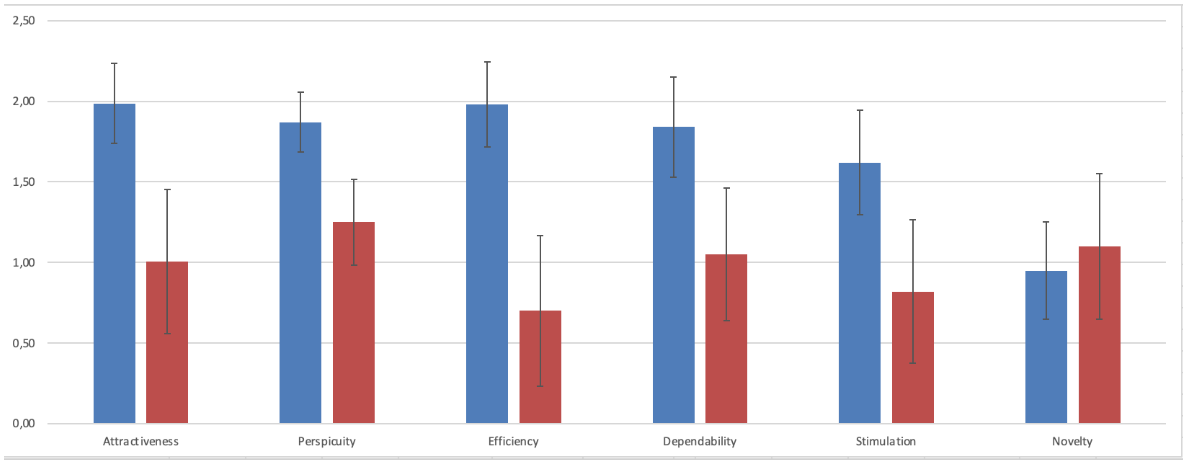


Figure 4.1: UEQ results for the test tool (blue) and the control tool (red)

of the test. When it comes to the attractiveness of the usage, the test tool scores an average of 1.99, while the control tool achieves almost a whole scaling point less, with just 1.01. Thus, the general satisfaction with the test tool proves to be higher.
For the perspicuity, the difference is less apparent, with the test tool achieving an average rating of 1.87 versus the 1.25 of the control tool. This falls in line with the observations made during the study, as both tools were generally easy to understand after a short explanation by the researcher.
The largest difference between the ratings of the tools is in the perceived efficiency, in which the test tool has an average rating of 1.98 and the control tool one of just 0.70. This was to be expected, as entering the prompt by themselves did significantly raise the time consumed to achieve results with the tool. Another factor could be the re-entering for prompts that was mentioned earlier, where in some cases the participants had to try two or more times to get a suitable output.
The dependability was rated with 1.84 on average for the test tool and 1.05 for the control tool. The biggest influence on this score most likely comes from the question about the predictability of the tool. As I mentioned earlier, there were a lot of cases where participants had to re-enter a different prompt after their original one did not yield a comment as output. Therefore, most of those participants rated the predictability to be on the lower side of the scale, which led to the difference in scores for the dependability dimension.
For the stimulation, the test group achieved an average result of 1.62 versus an average result of 0.82 for the control group. This could also be explained by the remark made for the dependability, as the longer workflow for the control tool most likely led to an impaired user experience in terms of stimulation.
The only dimension among the six that sees the control tool achieving a better result

than the test group is the dimension of novelty. This measures how innovative the tool felt for the participants. This result can most likely be explained by the simplicity of the test tool. The usage is relatively straightforward, and most of the heavy lifting, with the prompt and the generation of the comment, happens in the background without visual feedback for the user. The control tool, on the other hand, allows the user to experiment and offers functionalities beyond those of comment generation that are visible during the selection of the preferred use case. Therefore, one can derive that the control tool feels like it offers more and therefore achieved greater results in this category.

4.4.2 Survey Coding

Next, I coded the answers given by both groups to the text fields, asking about improvements and things that they enjoyed about each tool.
For the test tool, the main positive point mentioned by the participants was the easiness of the usage. Additionally, a lot of participants mentioned that the tool takes tedious work away from them in their daily workflows and is therefore very helpful. In terms of the output, most people rated the uniform comment structure positively and also mentioned the conciseness of the generated comments.
For the control group, some people mentioned the same points, with the ease of use and the helpfulness being the most mentioned aspects. Additionally, one participant mentioned that the tool helps take away personal bias when writing comments, which is something that can be said for both tools.
In terms of negative comments and change requests for the test tool, participants mainly mentioned the slow generation time for the comments. Apart from some minor changes requested, such as a progress indicator, some participants asked for more options for comment types and a better adjustment in terms of explanation depth for longer functions. This was most likely mentioned because the test tool kept the comments for both code snippets at about the same length, even though the second snippet required a more detailed documentation in the eyes of some participants. One person mentioned a perceived security risk from using the tool for internal company code, as it uses a non open-source LLM.
For the control tool, the negative feedback mainly mentioned the additional output generated, besides the comment. In combination with having to copy and paste the comments into the code files, this led to an impaired user experience, as mentioned earlier. Additionally, participants did not enjoy writing the prompts themselves, due to the longer time it took to get the output.

5 Discussion

Deep vs. general understanding of functions. The results of this study highlight many interesting points. First, I find that prompts constructed following the few-shot paradigm do, in fact, yield significantly better results in the space of code comment generation than human-written prompts. This is most notable in the categories of readability and unnecessary information. For longer, more complicated code snippets, they additionally offer better results in the categories of usefulness and helpfulness. Based on the survey data and the observations made during the experiment, I can most likely attribute the better ratings to the length of the generated documentation.

Without any examples, or at the very least, a template to follow, GPT-4 is not able to assess the preferred length for the output. Therefore, it gives results that are generally too long to be considered as useful in a development context. It is, however, important to note that in certain scenarios, these longer comments might actually be helpful to improve the understanding of complicated functions, as mentioned by a few participants. This brings the question: which part of the term "understanding" is most important when it comes to comments for code snippets? Should they provide a quick overview of how a function works to allow people to continue working with them, or should they offer a deeper understanding? Depending on the answer to that question, the tool should probably offer various options of comment depth.

Difference in ratings. Next, I find that students and people with working experience in software development-related fields have quite a differing perception of the ratings of the comments. This could be attributed to a few factors, such as the fact that the sample size for the non-student group was smaller and that most of the interviews with developers were done in an on-site setting, versus the online interviews for the students. Still, the differences are quite large and potentially open up an interesting research direction.

Issues with LLMs. I find that the biggest weakness of the extension developed in this study is the amount of time needed for the comment to be generated. In a quick experiment after the study was over, I tracked the time needed for the output to be generated and returned to the user and found that on average, the process took about 4 seconds, which seemed too long for people. This is an issue that, of course, lies in the nature of LLMs, as they take time to produce their output. Another issue arises through the inherent randomness of the output. There were only a handful of cases in which either tool offered the exact same output for the same prompt. No amount of structuring in the engineered prompt could prevent that issue. Further research could thus explore an approach in which multiple outputs are generated, and the developer then chooses a preferred one. Though this would lead to a longer time for the generation of the comments, as well as the entire workflow, as people now have to additionally spend time choosing a Docstring.

Future Work. Next, future work could investigate further options to perfect the written prompt to GPT-4 and other models. There are recent studies that recommend more example input/output pairs to get better results for generated documentation [21]. While the ratings by the participants in our study suggest that in the context of GPT-4, three examples are enough, it would be interesting to see how the model performs with more input or even an entirely different approach to writing the prompt.

Apart from that, I propose that utilizing one of the many new open-source LLMs to run the tool locally would be the most interesting new research direction. This also follows an interesting remark made by one of the participants concerning potential security risks that come with the use of tools based on closed models, such as GPT-4. In a future iteration of the tool, I plan to offer participants the opportunity to adjust the prompt by sending few-shots of their own, though this opens the tool up to potential prompt injection attacks [52]. Additionally, developers should generally be cautious when sending entire functions of their code to closed models, as they cannot be sure what happens with the code [52]. Therefore, the future of productivity-enhancing tools for coding most likely lies in open-source LLMs, even though fine-tuning them comes with a plethora of new challenges in itself [53].

What remains for the developer. The more general question that remains following the study is what work will be left to the developers in terms of documenting code after using our tool and future iterations of it. The results of this study indicate that neither developers nor students are necessarily experts in the field of prompt engineering. Therefore, we can assume that the task of prompting LLMs should most likely be done by the vendors of tools, offering just slight adjustments as options for the users. Still, in its current state, it is clear that cross-checking the output is of utmost importance, no matter how well one prompts the LLM. Without knowledge of an entire codebase, the model can only assume what certain classes and other mentioned functions do, based on the naming. Improving the tool to be able to run it with entire files of functions that have dependencies between each other seems like a potential solution to that problem. But even then, one would not be able to guarantee correctness, as it would need the knowledge of an entire codebase. And even if LLMs someday in the future get to a point where one can assume that the output is definitely correct, selecting the right examples for the prompt or even just a template remains as an integral task. This is mostly due to the fact that every software project brings its own unique challenges, depending on the domain, the people working on the project, and even the timeline [7]. The comments generated for functions by my tool should, therefore, always follow a style that fits the preferences of each individual developer team. Hence, nudging an LLM into the right direction will most likely remain as a task for the team members.

6 Threats to Validity

6.1 Internal Validity

There are several internal threats of the study that need to be mentioned.
Firstly, it is important to note that generally, the writing of code documentation is a task that is done on code that has been written by the developers themselves. In the context of the controlled experiment, I was not able to create such an environment without significant impact on the rating dimensions proposed by Hu et al. [9]. Therefore, it is highly likely that the dimensions concerning the amount of unnecessary and missing information would be better measured in a field study, where the tools would be used on code written in a professional context. The trade-off was accepted because the dimensions of helpfulness and especially usefulness were deemed to be lost completely in the context of a field study, as it is most likely more difficult for developers to examine their own comments in terms of usefulness for other people. Nevertheless, I propose that a field study should be conducted in the future to gain robustness on the other mentioned dimensions.
Secondly, the standardized UEQ consists of 26 opposing adjective dimensions. During the creation of the survey for this study, the second dimension of the questionnaire, which is "understandable/not understandable," was not put in, due to an oversight error made by the interviewer. Therefore the UEQ results might be compromised in their full expressiveness. It is, however, important to note that after evaluating the UEQ, once with the middle value of the scale put in for the dimension, among all results for both groups, and once with omitting the dimension entirely, I did not find any significant difference to the end results. This is in line with the observations by the interviewer, as the majority of participants in both groups did not have any significant trouble with understanding how either tool worked.
Next, I note that all of the casual inferences made in this study are, of course, limited by the number of functions I used in the experiment. With only two functions being checked, it could very well be the case that the results of the survey would change to an unknown degree when giving participants more functions to test the tools on. This

is due to the fact that every software function is unique and might be understood to a different degree by every participant [7]. The control tool might perform better on certain, more complicated functions, as they require longer and more detailed comments to be understood perfectly.
Finally, it is important to note that the split for students and people working in software development-related fields diminished the group sizes. Therefore the results regarding the ratings of the snippets, in particular those by the 20 software developers, are less expressive. I would recommend at this point to conduct the study again, with more software developers and fewer students, or at least a more balanced group.

6.2 External Validity

In terms of potential external threats, it is important to note that LLMs in general generate at least slight variations of the answers for the same prompt, no matter how well it has been constructed, due to their inherent randomness [1]. Therefore, the conclusions that I have drawn from the results of the study might not be generalizable, as the outputs given by GPT-4 might not have been optimal. In a rather small scale study, with 50 participants, it was not possible to mitigate this risk adequately. Additionally, it is important to mention that I only tested one specific LLM in the context of this study.
Secondly, the entire study was conducted on one single project codebase. The three functions I used as the few-shots for the prompt were taken from the EuXFEL codebase. While I took into account the quality requirements for code comments, I cannot exclude the possibility that the few-shots were not optimal. Therefore, the output of the test tool could probably be improved further. This falls in line with the general risk of using only functions from the same codebase. The tool might give out radically different results for different projects.
17 out of the 20 recruited professionals were employed by EuXFEL. Due to their proficiency in Python, they were deemed well-qualified for the study. I acknowledge that the employer influences participants' skills.
In contrast to most other studies conducted on the space of documentation generation, I opted to only use human evaluation techniques, instead of metrics such as BLEU, METEOR, etc. While this was a design choice, I cannot rule out that the aforementioned metrics might yield more robust results than human evaluation.

7 Conclusion

The purpose of this thesis was to examine whether engineered prompts for the LLM GPT-4 yield better results than human-written ad-hoc prompts in the subspace of code comment generation. To that end, I conducted a controlled experiment with 50 participants from various software-related backgrounds and had them rate generated comments for two functions from the EuXFEL codebase.
I find that the test tool yielded significantly better results for the dimensions of readability and unnecessary information for both code snippets. It generally provided comments that followed the provided examples and the template in length and detail, while the control tool was too broad and unstructured in its output. This led to worse readability and a plethora of unnecessary additional information in the comment.
For the more complicated of the two functions, I find that participants also perceived the comment generated by our tool to be more useful for developers and more helpful in improving their understanding of the code snippet. Though it should be noted that participants in the control group actually outscored the participants in the test group in the questions answered correctly for the snippet after the usage of the tool. Thus, it is possible that the perceived improvement in understanding and the actual understanding were significantly different for the people in the control group.
Still, with this study, I add to the existing research for various other LLMs [7, 8] by concluding that prompt engineering does indeed enhance the quality of the output. It is important to note, though, that there was a clear distinction between the ratings given by the students and the non-students in our study, potentially indicating that working developers might not see the same benefits as students. This should be explored in future research that might go beyond the space of code documentation generation, as it could indicate different priorities for each group when it comes to writing and interpreting software in general.
Finally, I find that LLM based tools for software-related tasks seem to offer a greater user experience when the users do not have to write the prompts themselves, suggesting that industry practitioners prefer assistance with constructing ideal prompts. While writing prompts seems to give users a greater sense of innovation during the usage, it

also decreases the reliability and efficiency of the output.
I propose that the next step in the research of this topic should be the usage of an open-source LLM that runs locally, thus not only adding potentially faster response times but also an extra security layer, as well as more fine-tuned answers. It would be interesting to see if a model like that would outperform our tool on the task of comment generation, with or without fine-tuning, and also if prompt engineering itself is still important for locally run models.
Finally, I hope that my work inspires others to replicate this study in other settings and examine potential aspects to advance Artificial Intelligence (AI)-powered tool support for developers in various software-related domains.

Appendix

.1 Axios API call to the OpenAI API

```
async function generateComment(selectedText: string, prompt?: string):
    Promise<string> {
     try {
         const client = axios.create({
             headers: {
                 Authorization: 'Bearer <AUTH-TOKEN>',
             },
         });

         const params = {
             model: 'gpt-4',
             messages: [
                 {
                     role: 'system',
                     content: 'You are a programmer.'
                 },
                 {
                     role: 'user',
                     content: FEW_SHOTS + (prompt || DEFAULT_PROMPT).
    replace('{FUNCTION_CONTENT}', selectedText),
                 },
             ],
         };

         const response = await client.post(
             'https://api.openai.com/v1/chat/completions',
             params
         );

         return response.data.choices[0].message.content.trim();
     } catch (error) {
         console.error('OpenAI API request error:', error);
```

```
        throw error;
    }
}
```

.2 Karabo Python Functions

```
1. def all_equal(iterable):
        iterator = iter(iterable)
        try:
            first = next(iterator)
        except StopIteration:
            return True
        return all(first == x for x in iterator)
```

```
2. def string_from_vector_bool(data):
        return ",".join(str(int(i)) for i in data)
```

```
3. def _is_nonintegral_number(value):
        is_floating = isinstance(value, (numbers.Complex, np.inexact))
        is_integer = isinstance(value, numbers.Integral)
        return is_floating and not is_integer
```

```
4. def _parse_date(date):
        if date is None:
            date = Timestamp()
        if isinstance(date, Timestamp):
            date = date.toLocal()
        d = dateutil.parser.parse(date)
        if d.tzinfo is None:
            d = d.replace(tzinfo=dateutil.tz.tzlocal())
        return d.astimezone(dateutil.tz.tzutc())
        .replace(tzinfo=None).isoformat()
```

```
5. def dictToHash(d):
        h = Hash()
        for k, v in d.items():
            if isinstance(v, dict):
                h[k] = dictToHash(v)
            elif isinstance(v, (list, tuple)):
                if len(v) > 0 and isinstance(v[0], dict):
                    h[k] = HashList(dictToHash(vv) for vv in v)
                else:
                    h[k] = v
            else:
                h[k] = v
        return h
```

```
6. def hashToDict(h):
        d = dict()
        for k, v in h.items():
            if isinstance(v, Hash):
                d[k] = hashToDict(v)
            elif isinstance(v, (list, tuple)):
                if len(v) > 0 and isinstance(v[0], Hash):
                    d[k] = [hashToDict(vv) for vv in v]
                else:
                    d[k] = v
            else:
                d[k] = v
        return d
```

Bibliography

[1] H. Tian, W. Lu, T. O. Li, X. Tang, S.-C. Cheung, J. Klein, and T. F. Bissyandé, *Is chatgpt the ultimate programming assistant -- how far is it?*, 2023. DOI: 10.48550/arXiv.2304.11938.

[2] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza, “Software documentation issues unveiled”, in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, New York, NY, USA: IEEE, 2019, pp. 1199–1210. DOI: 10.1109/ICSE.2019.00122.

[3] I. Steinmacher, T. U. Conte, C. Treude, and M. A. Gerosa, “Overcoming open source project entry barriers with a portal for newcomers”, in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 273–284. DOI: https://doi.org/10.1145/2884781.2884806.

[4] I. Steinmacher, C. Treude, and M. A. Gerosa, “Let me in: Guidelines for the successful onboarding of newcomers to open source projects”, *IEEE Software*, vol. 36, no. 4, pp. 41–49, 2019. DOI: 10.1109/MS.2018.110162131.

[5] N. J. Kipyegen and W. P. Korir, “Importance of software documentation”, *International Journal of Computer Science Issues (IJCSI)*, vol. 10, no. 5, p. 223, 2013.

[6] P. W. McBurney, S. Jiang, M. Kessentini, N. A. Kraft, A. Armaly, M. W. Mkaouer, and C. McMillan, “Towards prioritizing documentation effort”, *IEEE Transactions on Software Engineering*, vol. 44, no. 9, pp. 897–913, 2017. DOI: https://doi.org/10.1109/TSE.2017.2716950.

[7] T. Ahmed and P. Devanbu, “Few-shot training llms for project-specific code-summarization”, in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22, Rochester, MI, USA: Association for Computing Machinery, 2023, ISBN: 9781450394758. DOI: 10.1145/3551349.3559555. [Online]. Available: https://doi.org/10.1145/3551349.3559555.

[8] T. Ahmed, K. S. Pai, P. Devanbu, and E. T. Barr, *Improving few-shot prompts with relevant static analysis products*, 2023. DOI: 10.48550/arXiv.2304.06815.

[9] X. Hu, Q. Chen, H. Wang, X. Xia, D. Lo, and T. Zimmermann, “Correlating automated and human evaluation of code documentation generation quality”, *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, Jul. 2022, ISSN: 1049-331X. DOI: `10.1145/3502853`. [Online]. Available: `https://doi.org/10.1145/3502853`.

[10] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, H. Nori, H. Palangi, M. T. Ribeiro, and Y. Zhang, *Sparks of artificial general intelligence: Early experiments with gpt-4*, 2023. DOI: `10.48550/arXiv.2303.12712`.

[11] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need”, *Advances in neural information processing systems*, vol. 30, 2017.

[12] J. Kocoń, I. Cichecki, O. Kaszyca, M. Kochanek, D. Szydło, J. Baran, J. Bielaniewicz, M. Gruza, A. Janz, K. Kanclerz, *et al.*, “Chatgpt: Jack of all trades, master of none”, *Information Fusion*, p. 101 861, 2023. DOI: `https://doi.org/10.1016/j.inffus.2023.101861`.

[13] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, *Large language models for software engineering: A systematic literature review*, 2023. DOI: `10.48550/arXiv.2308.10620`.

[14] R. OpenAI, *Openai (2023)*, 2023. DOI: `https://doi.org/10.48550/arXiv.2303.08774`.

[15] C. Ebert and P. Louridas, “Generative ai for software practitioners”, *IEEE Software*, vol. 40, no. 4, pp. 30–38, 2023. DOI: `10.1109/MS.2023.3265877`.

[16] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, “Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing”, *ACM Computing Surveys*, vol. 55, no. 9, Jan. 2023, ISSN: 0360-0300. DOI: `10.1145/3560815`. [Online]. Available: `https://doi.org/10.1145/3560815`.

[17] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, *A prompt pattern catalog to enhance prompt engineering with chatgpt*, 2023. DOI: `10.48550/arXiv.2302.11382`.

[18] R. L. L. IV, I. Balažević, E. Wallace, F. Petroni, S. Singh, and S. Riedel, *Cutting down on prompts and parameters: Simple few-shot learning with language models*, 2021. DOI: `10.48550/arXiv.2106.13353`.

[19] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners", vol. 33, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., pp. 1877–1901, 2020. [Online]. Available: `https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf`.

[20] J. Y. Khan and G. Uddin, "Automatic code documentation generation using gpt-3", in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–6. DOI: `https://doi.org/10.1145/3551349.3559548`.

[21] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, and X. Liao, "Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning", 2024.

[22] S. Rai, R. C. Belwal, and A. Gupta, "A review on source code documentation", *ACM Transactions on Intelligent Systems and Technology*, vol. 13, no. 5, Jun. 2022, ISSN: 2157-6904. DOI: `10.1145/3519312`. [Online]. Available: `https://doi.org/10.1145/3519312`.

[23] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods", in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10, Antwerp, Belgium: Association for Computing Machinery, 2010, pp. 43–52, ISBN: 9781450301169. DOI: `10.1145/1858996.1859006`. [Online]. Available: `https://doi.org/10.1145/1858996.1859006`.

[24] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context", in *Proceedings of the 22nd International Conference on Program Comprehension*, ser. ICPC 2014, Hyderabad, India: Association for Computing Machinery, 2014, pp. 279–290, ISBN: 9781450328791. DOI: `10.1145/2597008.2597149`. [Online]. Available: `https://doi.org/10.1145/2597008.2597149`.

[25] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization", in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10, Cape Town, South Africa: Association for Computing Machinery, 2010, pp. 223–226, ISBN: 9781605587196.

DOI: 10.1145/1810295.1810335. [Online]. Available: https://doi.org/10.1145/1810295.1810335.

[26] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation", in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 200–210, ISBN: 9781450357142. DOI: 10.1145/3196321.3196334. [Online]. Available: https://doi.org/10.1145/3196321.3196334.

[27] U. Alon, O. Levy, and E. Yahav, "Code2seq: Generating sequences from structured representations of code", 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX.

[28] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network", in *Proceedings of the 28th International Conference on Program Comprehension*, ser. ICPC '20, Seoul, Republic of Korea: Association for Computing Machinery, 2020, pp. 184–195, ISBN: 9781450379588. DOI: 10.1145/3387904.3389268. [Online]. Available: https://doi.org/10.1145/3387904.3389268.

[29] S. Gao, C. Gao, Y. He, J. Zeng, L. Nie, X. Xia, and M. Lyu, "Code structure–guided transformer for source code summarization", *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, Feb. 2023, ISSN: 1049-331X. DOI: 10.1145/3522674. [Online]. Available: https://doi.org/10.1145/3522674.

[30] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning", in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18, Montpellier, France: Association for Computing Machinery, 2018, pp. 397–407, ISBN: 9781450359375. DOI: 10.1145/3238147.3238206. [Online]. Available: https://doi.org/10.1145/3238147.3238206.

[31] W. Wang, Y. Zhang, Z. Zeng, and G. Xu, "Trans3: A transformer-based framework for unifying code summarization and code search", 2020. DOI: 10.48550/arXiv.2003.03238.

[32] S. Aljumah and L. Berriche, "Bi-lstm-based neural source code summarization", *Applied Sciences*, vol. 12, no. 24, 2022, ISSN: 2076-3417. DOI: 10.3390/app122412587. [Online]. Available: https://doi.org/10.3390/app122412587.

[33] S. Birari and S. Bhingarkar, "Using artificial intelligence in source code summarization: A review", *Recent Trends in Intensive Computing*, vol. 39, pp. 256–262, 2021. DOI: 10.3233/APC210203.

[34] Y. Gao and C. Lyu, “M2ts: Multi-scale multi-modal approach based on transformer for source code summarization”, in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ser. ICPC '22, Online: Association for Computing Machinery, 2022, pp. 24–35, ISBN: 9781450392983. DOI: 10.1145/3524610.3527907. [Online]. Available: https://doi.org/10.1145/3524610.3527907.

[35] W. Sun, C. Fang, Y. You, Y. Miao, Y. Liu, Y. Li, G. Deng, S. Huang, Y. Chen, Q. Zhang, H. Qian, Y. Liu, and Z. Chen, *Automatic code summarization via chatgpt: How far are we?*, 2023. DOI: 10.48550/arXiv.2305.12865.

[36] R. Durelli, V. Durelli, R. Bettio, D. Dias, and A. Goldman, “Divinator: A visual studio code extension to source code summarization”, in *Anais do X Workshop de Visualização, Evolução e Manutenção de Software*, Online: SBC, 2022, pp. 1–5. DOI: 10.5753/vem.2022.226187. [Online]. Available: https://sol.sbc.org.br/index.php/vem/article/view/22320.

[37] Github. (2023). “Copilot”. [Online; retrieved on 28.12.2023], [Online]. Available: https://github.com/features/copilot.

[38] Zhang Renyang. (2023). “ChatGPTExtension”. [Online; retrieved on 31.10.2023], [Online]. Available: https://marketplace.visualstudio.com/items?itemName=zhang-renyang.chat-gpt.

[39] European-XFEL. (2023). “Karabo”. [Online; retrieved on 24.08.2023], [Online]. Available: https://github.com/European-XFEL/Karabo/.

[40] S. Kapil, *Clean Python*. Berkeley, CA, USA: Apress, 2019. DOI: 10.1007/978-1-4842-4878-2. [Online]. Available: https://doi.org/10.1007/978-1-4842-4878-2.

[41] Team UEQ. (2023). “UEQ”. [Online; retrieved on 24.08.2023], [Online]. Available: https://www.ueq-online.org.

[42] OpenAI. (2023). “OpenAIPromptGuide”. [Online; retrieved on 22.12.2023], [Online]. Available: https://platform.openai.com/docs/guides/prompt-engineering/six-strategies-for-getting-better-results.

[43] P. Rani, A. Blasi, N. Stulova, S. Panichella, A. Gorla, and O. Nierstrasz, “A decade of code comment quality assessment: A systematic literature review”, *Journal of Systems and Software*, vol. 195, p. 111 515, 2023, ISSN: 0164-1212. DOI: 10.1016/j.jss.2022.111515. [Online]. Available: https://doi.org/10.1016/j.jss.2022.111515.

[44] Microsoft. (2023). “Visual Studio Code”. [Online; retrieved on 24.08.2023], [Online]. Available: https://code.visualstudio.com/api/get-started/your-first-extension.

[45] O. Alhadreti and P. Mayhew, “Rethinking thinking aloud: A comparison of three think-aloud protocols”, in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI '18, Montreal, Canada: Association for Computing Machinery, 2018, pp. 1–12, ISBN: 9781450356206. DOI: 10.1145/3173574.3173618. [Online]. Available: https://doi.org/10.1145/3173574.3173618.

[46] B. Derrick, D. Toher, and P. White, “Why welch's test is type i error robust”, *The Quantitative Methods for Psychology*, vol. 12, no. 1, pp. 30–38, 2016. DOI: 10.20982/tqmp.12.1.p030. [Online]. Available: https://doi.org/10.20982/tqmp.12.1.p030.

[47] N. Nachar, “The mann-whitney u: A test for assessing whether two independent samples come from the same distribution”, *Tutorials in Quantitative Methods for Psychology*, vol. 4, no. 1, pp. 13–20, 2008. DOI: 10.20982/tqmp.04.1.p013. [Online]. Available: https://doi.org/10.20982/tqmp.04.1.p013.

[48] T. S. community. (2024). “Statistical function”, The SciPy community, [Online]. Available: https://docs.scipy.org/doc/scipy/reference/stats.html (visited on 01/04/2024).

[49] B. Laugwitz, T. Held, and M. Schrepp, “Construction and evaluation of a user experience questionnaire”, in *HCI and Usability for Education and Work*, A. Holzinger, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 63–76, ISBN: 978-3-540-89350-9. DOI: 10.1007/978-3-540-89350-9_6. [Online]. Available: https://doi.org/10.1007/978-3-540-89350-9_6.

[50] M. A. Cascio, E. Lee, N. Vaudrin, and D. A. Freedman, “A team-based approach to open coding: Considerations for creating intercoder consensus”, *Field Methods*, vol. 31, no. 2, pp. 116–130, 2019. DOI: 10.1177/1525822X19838237. [Online]. Available: https://doi.org/10.1177/1525822X19838237.

[51] N. Mohd Razali and B. Yap, “Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests”, *Journal of Statistical Modeling and Analytics*, vol. 2, no. 1, pp. 21–33, 2011.

[52] S. Abdelnabi, K. Greshake, S. Mishra, C. Endres, T. Holz, and M. Fritz, “Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection”, in *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, ser. AISec '23, Copenhagen, Denmark: Association for

Computing Machinery, 2023, pp. 79–90, ISBN: 9798400702600. DOI: 10.1145/3605764.3623985. [Online]. Available: https://doi.org/10.1145/3605764.3623985.

[53] J. Kaddour, J. Harris, M. Mozes, H. Bradley, R. Raileanu, and R. McHardy, *Challenges and applications of large language models*, 2023. DOI: 10.48550/arXiv.2307.10169. [Online]. Available: https://doi.org/10.48550/arXiv.2307.10169.

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Hamburg, den 29. Januar 2024

[redacted]

Hans-Alexander Christian Kruse

Erklärung zur Veröffentlichung

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Hamburg, den 29. Januar 2024

[redacted]

Hans-Alexander Christian Kruse