Prompt Engineering with Requirements in GitHub Copilot

Exploring the ways that requirements can help Copilot with accurately solving self-admitted technical debt found inside GitHub repositories

Master's Thesis in Computer science and engineering

Anwarr Shiervani

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's Thesis 2025

Prompt Engineering with Requirements in GitHub Copilot

Exploring the ways that requirements can help Copilot with accurately solving self-admitted technical debt found inside GitHub repositories

Anwarr Shiervani

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Prompt Engineering with Requirements in GitHub Copilot
Anwarr Shiervani

Supervisor: Linda Erlenhov, Department of Computer Science and Engineering
Examiner: Rebekka Wohlrab, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Anwarr Shiervani
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Practices within requirements engineering are essential for the success of software projects. In particular, document analysis enables the elicitation of requirements without engaging directly with the stakeholders. This thesis presents a method of scraping documents from English open-source software repositories hosted on GitHub and extracting requirements from these documents. This is done in order to supply GitHub Copilot with more context when prompting it to repay self-admitted technical debt expressed through TODO comments. Three different prompt templates were created to evaluate this approach. The first template had no requirements in the prompt, the second template had some relevant requirements, and the third one had all the relevant requirements. After prompting Copilot, prompts from the first template were shown to the best at repaying debt – amounting to a repayment rate of 72%. However, they also accounted for the most repayments that did not conform to all the requirements. While prompts from the third template showed the best results in regard to this, they displayed a lower debt repayment rate of 64%. Finally, it is noted that self-admitted debt is not truly repaid unless done in a way that conforms to the relevant requirements. As such, there is a need for future research to expand upon this particular problem through the development of a robust framework.

Keywords: Software Development, Artificial Intelligence, Self-Admitted Technical Debt, Requirements Engineering, GitHub Copilot, Prompt Engineering, Requirements Elicitation, GitHub, Open-Source Software, Natural Language Processing

Acknowledgements

I would like to thank my supervisor Linda Erlenhov, and my examiner Rebekka Wohlrab, for giving me constructive feedback as I progressed through my master thesis.

Anwarr Shiervani, Gothenburg, June 2025

Contents

List of Figures

List of Tables

1

Introduction

1.1 Problem Description

Software development driven by artificial intelligence (AI) is on the rise. A survey conducted by Stack Overflow [1] found that 76% of the respondents were positive about using AI in their workflows. For this purpose, software developers have various tools at their disposal. Davila et al. [2] found that industry practitioners primarily made use of ChatGPT [3] and GitHub Copilot [4] for AI-assisted development. In particular, these two tools use large language models (LLMs) to fulfill their tasks.

With the use cases of AI-driven coding assistants in mind, it is crucial to consider their potential issues when used in practice. Take code comments that are annotated with the tag “TODO”. According to Potdar and Shihab [5], these TODO comments contain indications of self-admitted technical debt (SATD). OBrien et al. [6] sampled such comments from open source software (OSS) repositories hosted on GitHub [7], and used them in prompts to Copilot to repay the corresponding instances of SATD. Findings from their experiments showed interesting results. For example, Copilot successfully repaid SATD, expressed in roughly 10% more TODO comments if the word 'TODO' was removed from the comment before inserting it in the prompt. However, while they could repay roughly 32% of all SATD instances using simpler techniques, they did not consider whether the generated code conformed to the intended functionality. These aspects are especially problematic when a survey, conducted by Sergeyuk et al. [8], found 49% of its respondents willing to delegate refactoring tasks (which are meant to get rid of technical debt) to AI coding tools.

Hofmann and Lehner [9] found that the proper execution of requirements engineering (RE) processes is a common factor in the success of software projects. With that in mind, this thesis aims to repay instances of SATD, found in TODO comments from a dataset published by OBrien et al. [10], through the inclusion of requirements in the prompt to GitHub Copilot. The goal of eliciting and including relevant requirements in prompts to Copilot, is that the generated code ends up functioning closer to the needs of different project stakeholders.

Overall, the findings of this thesis are thought to be useful to OSS developers who use AI-driven coding assistants (e.g., GitHub Copilot) when contributing to repositories on GitHub. For example, such developers can use the scripts in Section A.1 (guided by Chapter 4) to see how much the inclusion of relevant requirements can help Copilot with the accurately repaying SATD expressed through a TODO comment.

1.2 Research Questions

The four research questions (RQs) are as follows:

- **RQ 1:** *To what extent can three groups of requirement documents, related to TODO comments, be automatically elicited from their GitHub repositories?*

  This RQ aims to assemble three groups of requirement documents from all GitHub repositories, in which the TODO comments are found. A group represents either a pull request, an issue, or a commit on the repository pertaining to a TODO comment. Due to the limited scope of the thesis, no more than three groups were gathered. Requirement documents, which are stored within the groups, denote user comments that are found to indicate the presence of one or more requirements. These documents should be in some way related to a TODO comment. If at least three groups of requirement documents cannot be found for a certain TODO comment, then the TODO comment is not considered for answering the remaining RQs.

- **RQ 2:** *To what extent does Copilot repay SATD reported by TODO comments, by generating code that conforms to requirements in all groups of requirement documents?*

  The aim of this RQ is to evaluate Copilot's ability to repay instances of SATD, expressed in the TODO comments, in accordance with all the requirements. No requirements are specified in the prompts here, instead, Copilot will have to figure out these requirements by itself when tasked with the repayment of SATD.

- **RQ 3:** *How effective is Copilot's repayment of SATD, while conforming to requirements in all groups of requirement documents, if only the first group was included in the prompt?*

  This RQ is similar to RQ 2. However, it will include one group of requirement documents in the prompt to Copilot. Since all groups are sorted according to some of their attributes (e.g., their relevance to a certain TODO comment), only the very first group of requirement documents will be used for this RQ. The aim here is to see whether Copilot can figure out requirements from the other two groups by itself and repay SATD according to requirements in all three groups. It could also be the case that the other groups contain useless requirements, therefore, there is no need for Copilot to even consider these in the first place.

- **RQ 4:** *How effective is Copilot's repayment of SATD, while conforming to requirements in all groups of requirement documents, if all these groups were* *included in the prompt?*

In contrast to RQ 3, prompts generated for this RQ will contain requirement documents from all the three groups elicited in RQ 1. The point of answering this RQ is to see whether Copilot needs access to all groups of requirement documents, in order to repay SATD in a way that aligns more closely with the needs of project stakeholders.

2

Background

2.1 Requirements Engineering (RE)

Ensuring cohesiveness in the planning, development, and maintenance of software projects can be problematic. By combining various practices in the field of RE, organizations may overcome this problem [9].

Software requirements are specifications for how a client expects a piece of software to be designed and behave. Exactly which aspect(s) of the application's functionality these requirements aim to specify depends on the type of requirement in question. There are two common types of requirements: functional and non-functional.

Lauesen [11, p. 14–15], functional requirements (FR) go into detail about input/output transformations performed by an application. They can be specified in various ways: context diagrams (the interfaces that parts of the application should support), task descriptions (which actions are performed by the user versus the application, to complete a particular task), etc. In contrast to FRs, Lauesen [11, p. 15] argues that non-functional requirements (NFR) instead focus on describing quality attributes that the application shall fulfill. Just like FRs, they can be specified in many ways. Examples are: quality grids (assigning different priorities to different qualities), quality factors (conforming to qualities described in predefined lists), etc.

2.1.1 Elicitation

Software requirements must be gathered, and requirements elicitation is used for this. Requirements may be elicited through several practices. Examples are: stakeholder interviews, document analysis, and so on [11, p. 338]. Davey and Parker show that the elicitation process can be challenging regardless of the underlying practice used [12].

There are many reasons why the elicitation process might not go as initially planned. For example, an employee in the client organization may be inclined to sabotage the process due to their motives. This is why the requirements engineer has to be skilled in navigating such scenarios. Ebert and Wieringa [13, p. 5–6] suggests that requirement engineers should combine multiple elicitation practices to overcome their respective weaknesses when used by themselves. Therefore, combining two elicitation practices is particularly useful for this thesis: document analysis and stakeholder analysis.

Document analysis is helpful because it enables the elicitation of requirements without direct interaction with stakeholders. Instead, the practice involves technical documents that are manually or automatically crawled for requirements. Gobov and Huchenko [14] found this practice relatively common. 87% of respondents in their study used document analysis to elicit requirements from organizational documents. However, document analysis does not come without its problems. Identifying the exact type of requirements (i.e., FR or NFR) found in relevant sources during document analysis can be rather cumbersome if there are many documents to consider. Thankfully, there are various techniques that one may leverage to overcome such difficulties.

Lim et al. [15] mention some standard techniques discovered in literature. Rule-based classification can be particularly useful, as it can automatically elicit requirements from text documents. Johann et al. [16] found this could be done using natural language processing (NLP). In doing so for 10 different software projects, their technique demonstrated an accuracy of 87% for the correct and automatic elicitation of requirements.

Finally, according to Lauesen [11, Sec. 8.2], stakeholder analysis can help highlight the concerns of people who are invested in the software project. This analysis includes identifying their utmost needs, the risks they see with the project, etc. Pacheco and Tovar [17] argue that answering these questions only aims to cover one of the three aspects in the analysis process. The remaining two aspects look at stakeholder interactions (e.g., problem alignment) and the stakeholders themselves (e.g., their power).

2.2 Technical Debt (TD)

While it is certainly possible, not every software developer writes flawless code. It is common to expedite the software development process at the cost of sacrificing certain qualities of the source code. Doing so causes an instance of a phenomenon called technical debt (TD) [18]. De Almeida et al. [19] found that the primary reasons for settling with TD included businesses valuing the creation of new features over the quality of source code and enforcing strict deadlines for the application's delivery. Indeed, Rios et al. [20] conducted interviews with members of software development teams and found that issues during the project planning proved to be a prevalent cause for instances of TD. The interviews also showed that team members lacking adequate knowledge (e.g., technical expertise) played a role in causing TD.

TD can be divided into numerous categories. Alves et al. [21] lists and describes many of these, along with the indicators that can help identify them. For example, design debt refers to code that violates the concepts taught in object-oriented programming (OOP). An example is the god class, which Vaucher et al. [22] identify as a class that centralizes too much of the logic within the application. Multiple categories of TD may share one or more indicators. In the case of code debt, which refers to the decreased ability to comprehend a particular piece of code, god classes are also deemed to be an indicator.

In the long run, Ramač et al. [23] discovered that the presence of TD made it even harder to keep up with project deadlines. Its presence also resulted in incomprehensible code, which undoubtedly harms the application's maintainability. To overcome the implications of TD, one has to know how to identify, manage, and eventually repay the debt. Identifying TD is at least as crucial as its repayment, considering that project managers may not see the immediate need to repay TD. For the same reason, managing TD is also an integral step in paving the road for its repayment.

2.2.1 Identification and Management

The identification of TD can happen either manually or automatically. As for the manual identification of TD, Tan et al. [24] annotated sentences inside issue trackers that they deemed to indicate some form of TD. The problem with that particular method is that this process may be infeasible for larger software projects. Biazotto et al. [25] found that the automatic identification of TD is possible through specialized tools. These tools commonly use code metrics, such as the lines of code in a function, to identify TD in source code repositories.

Once that TD has been identified, it should be appropriately managed if there is no intention of its immediate repayment. Management of TD primarily involves keeping track of it, e.g., position in the source code. However, as shown by Alves et al. [26], certain techniques can help an optimal estimation of when its repayment should be done (e.g., during periods with relatively low workloads).

2.2.2 Repayment

The manual repayment of TD involves a developer fixing the issue related to the TD. Depending on the type of TD at hand, a relevant repayment practice is chosen. Standard practices to repay debt involve refactoring the code and the application's design and revising parts of the documentation (e.g., code comments).

Repayment of TD is avoided mostly for non-technical reasons. One prevalent cause is a lack of long-term interest in certain qualities, e.g., maintainability of code [27]. Since manually repaying TD may prove to be cumbersome, having it repaid automatically might be the preferred option. Exactly how the automatic repayment is done depends on the TD at hand. In the case of code debt, Johnson et al. [28] mention some automated static analysis tools (ASAT) that can automatically refactor pieces of code to get rid of the debt. Szőke et al. [29] created and tested such a tool in practice. They found that their ASAT solved instances of code debt in the thousands.

2.2.3 Self-Admitted Technical Debt (SATD)

While all sorts of TD may manifest as an unintentional degradation of software quality, there are instances where TD is purposefully caused and subsequently acknowledged by a developer. Potdar and Shihab [5] define this type of TD as self-admitted technical debt (SATD). They found that code comments served as sources from which instances of SATD could be extracted. Furthermore, their investigation of the presence of SATD and its lifecycle in a few popular software repositories yielded some interesting results. Among other things, the repayment of SATD in those repositories varied from 26.3% to 63.5%. This shows that the prioritization of resolving SATD can differ significantly between different software repositories.

There does not seem to be a significant variation in the types of TD that instances of SATD commonly concern themselves with. Maldonado and Shihab [30] analyzed code comments in software repositories and found five main types of TD in the comments that express SATD. The most common type of TD was found to be design debt. As mentioned in Section 2.2, design debt may manifest through one or more god classes. Other types of debts, such as documentation debt, were not found anywhere as prevalent in the studied repositories.

The detection of SATD in software repositories is possible through various techniques. Sutoyo and Capiluppi [31] have conducted a thorough literature review to identify some of these and measured their accuracy through experiments. One of those techniques included NLP (by looking for e.g., annotation tags, textual patterns). NLP was found to lack when identifying a wide variety of SATD types. However, annotation tags (i.e., locating code comments containing tags such as “TODO”) demonstrated a promisingly low number of false positives for detecting SATD in the aforementioned experiments.

As for the repayment of SATD, a few notable factors seem to affect it. Li et al. [32] found that the source from which the SATD is found seemingly influences the speed at which it is resolved. In one of the two software repositories they analyzed in the paper, they found that instances of SATD found in issue trackers were resolved 157 days faster on average compared to the ones found in code comments. Additionally, the authors found that developers who cause instances of SATD are also the ones who resolve it in the end.

2.3 Open-Source Software (OSS)

According to the HPE Glossary [33], software projects belong to one of the two categories: closed-source software and open-source software (OSS). Owners of closed-source software projects are typically found to impose heavy restrictions upon the rights of their end users, while authors of OSS generally aim to do the opposite. Some characteristic traits of OSS repositories include accepting contributions to the code (e.g., through pull requests [34]) and a public issue tracker.

Capturing the motivations of OSS developers is essential for the purpose of understanding the dynamics behind their interactions within OSS communities. A survey conducted by Hars and Ou [35] had participants from OSS communities explain their reasons for engaging in OSS. About 71% of the respondents participated in open-source software development (OSSD) as a way to improve upon their programming knowledge. The survey also hinted towards a correlation between the respondents' occupation and the type of motivations that they had. Student contributors seemed to be driven more by internal incentives than the salaried contributors. That is, students engage in OSS simply because they feel that it is satisfying.

However, despite the motivations that serve to drive OSS contributors, some projects in OSSD end up failing for one or more reasons. Coelho and Valente [36] identified some of those reasons through a survey with maintainers of various OSS repositories that were seemingly abandoned. Many of the reasons were related to certain qualities of the project, such as its maintainability. In particular, one respondent pointed out how certain architectural debt in the project caused them to stop maintaining it. Finally, as for how to pave the road for success, the authors found that the presence of a guideline document for contributions seemed to affect the likelihood of an OSS project succeeding.

2.3.1 GitHub

One of the biggest platforms for hosting OSS projects is GitHub [7]. As reported by the company in 2022 [37], it had around 94 million registered contributors and hosted about 413 million contributions in total. To host the repositories on the platform, it uses the Git version control system [38]. Version control systems help track changes to the source code through commits. Commits are data objects that contain metadata surrounding specific changes that were made [39], and each commit is uniquely identified by its SHA-1 reference [40]. Some of the metadata that commits contain includes the name of the author who made the changes, the date these changes were made, and so on.

Of course, GitHub is more than just a frontend for Git. The platform supports functionality that goes well beyond Git's capabilities. Two significant examples of such extensions include the issue tracker and pull requests. The issue tracker is used for making other contributors aware of the issues in a project by keeping information related to a particular issue all in one place [41]. Despite its name, issues can be used for things beyond bug reports. For example, a contributor may post an issue to keep track of an idea they had in mind for the project. As for pull requests, these are used for proposing changes to the software repository in question [34]. The point of posting pull requests is, more or less, the same as posting issues. That is, both of these functions foster community engagement. In fact, the GitHub API [42] considers pull requests to be a subclass of issues. In the case of a pull request making changes to some code, other contributors may chime in with their opinions regarding those changes through a pull request review. Also, comments from contributors can be found in more places than just inside of issues and pull requests. Commits serve as yet another source since contributors on GitHub are given the option of commenting on them [43].

2.3.1.1 Data Mining

Mining information from comments found on GitHub repositories can be handy for conducting OSSD research. In particular, Kalliamvakou et al. [44] found that the trend of mining GitHub for information is becoming more common in the research community. Thus, there is a need to explain the pitfalls of using this technique as the basis for a study. The authors revealed some of these pitfalls by analyzing data from surveys, interviews, and a dataset of GitHub repositories. Firstly, they found that repositories that serve as forks do not represent entire software projects alone. Instead, the main repository and its fork(s) represent the software project in its entirety. Having this in mind while mining GitHub repositories for data will give a better picture of the software project and its related community activities. Secondly, they found that most repositories are not used for collaboration with other contributors on GitHub. In fact, about two-thirds of the repositories in the dataset were only committed to by one contributor and, therefore, seemingly only intended for personal use.

Another problematic aspect with mining information from GitHub is the author of a particular comment, and whether they are human (as one would otherwise believe them to be). This is a real problem because, as Wessel et al. [45] showed through their manual analysis of activities in OSS repositories, the use of software tools (known as “bots”) for the automation of certain actions in OSSD is becoming more prevalent. Among other activities, they found these bots to be used for reviewing pull requests. Dependabot [46] is an example of an account on GitHub that is (explicitly) marked as a bot. This bot is, among other things, used to cover updates of repository dependencies by automatically posting pull requests for these. When it came to inactive repositories with Dependabot updates enabled, that particular feature can manifest itself through a great deal of spam. This was the case until GitHub presented a change [47], which made it so that Dependabot is disabled based on some inactivity criteria. However, as the change from GitHub was applied proactively after `2023-01-12`, it does not affect the state of pull requests submitted by the bot in the past.

However, Golzadeh et al. [48] found that the GitHub API does not always manage to explicitly distinguish bots as truly being bots; particularly in cases which these are operating through a “machine user account” [49]. The authors suggest using specialized classification tools to account for this issue. Recently, Chidambaram et al. [50] created and evaluated one such tool on a dataset of bots and humans on GitHub. The tool performed quite well compared to similar tools made in the past, using far fewer requests while doing so; demonstrating a remarkably efficient use of the constrained GitHub API rate limit [51]. However, its execution is dependent on certain events that are performed by the accounts that are being analyzed. According to GitHub [52], events that were performed more than 90 days in the past cannot be retrieved through their API. This means that the tool is useless for detecting bots with no recent events logged and those that operate sporadically. Furthermore, while it demonstrated outstanding performance, it was still not found to be perfectly accurate. This is detrimental for OSSD research, as one false positive could potentially result in the exclusion of many human-written documents.

Finally, it is important to note that the GitHub Terms of Service [53, Sec. D.6] states that all content on a certain GitHub repository is subsequently licensed under the repository license once posted on one of its pages. If a repository lacks an attached license, then no rights are given for the work (e.g., modifying it) except for simply viewing it [54]. With that in mind, there is a risk of using pieces of code and documents from these repositories in an unauthorized manner. A particular post by a user on GitHub [55] mentioned that the GitHub API endpoint for retrieving the repository license [56] fails to properly do so in the case of dual-licensing. Therefore, solely relying on the GitHub API for this is out of the question. Licensee [57] seemed like a promising tool to use for this purpose. However, according to the developers themselves [58], the tool cannot accurately find licenses specified in READMEs. Therefore, checking the license(s) that a particular repository falls under involves work that has to be undertaken manually.

2.3.2 Requirements in Open-Source Software

There are some deviations in how OSS developers practice RE, at the very least, from how industry practitioners do it. Kuriakose and Parsons [59] showed that the majority of RE practices in OSSD were found to be performed informally. Furthermore, Scacchi [60] argued that the specification of requirements in OSSD was largely decentralized; they were seldom found through a single artifact. Instead, requirements for OSS projects could be found in artifacts all over the Internet (and even outside of it).

As mentioned in Section 2.1.1, rule-based document analysis is useful for automatically eliciting requirements. Research shows that there are ways of performing this task with OSS projects. Vlas and Robinson [61] developed and demonstrated the use of a framework, named RCNL, intended for this particular purpose. The authors specify rules at each framework's five levels to locate and classify requirements in documents. Levels 0 to 1 are said to be common to all NLP frameworks, as rules at these levels aim to identify tokens (e.g., words, delimiters) and parts of speech (e.g., adjectives, verbs) respectively. As for the other levels, containing rules unique to the RCNL framework, the authors found that qualifier-rules at level 2 were the ones that actually ended up identifying the requirements.

2.3.3 Technical Debt in Open-Source Software

As with any other software repository, TD is also present inside OSS repositories. Potdar and Shihab [5] analyzed code comments inside several popular OSS repositories to identify the presence of SATD. They used a manual approach to identify instances of SATD, and later presented a list of textual patterns [62] that they found the SATD to follow. It was noted that up to a third of the source files within all the repositories indicated one or more instances of SATD. Contrary to how deadlines incurred TD in software development companies (as previously discussed in Section 2.2), SATD in OSSD was not found to be caused by pressure to push for release.

Regarding the repayment of TD in OSSD, Amanatidis et al. [63] discovered some interesting patterns. Firstly, repository contributors were almost 4 times more likely to prioritize the repayment of TD for certain quality factors than others (e.g., testability over maintainability). Secondly, users who had not previously contributed to the project were 3 times more willing to get rid of some TD than the actual contributors. The reluctance of these contributors, when it came to accepting requests for the repayment of some TD they may have introduced, was said to potentially be related to community-specific beliefs and practices. On the other hand, Tan et al. [64] also found that OSS contributors who caused a certain instance of TD ended up being the ones to identify and fix it themselves. This shows that the nature of TD repayment is rather complex when it comes to OSSD.

2.4 AI Coding Assistants

Large language models (LLMs) consist of neural networks that contain relationships between different textual entities (e.g., sentences) [65]. Every LLM has its specific area of application that it excels in, and this area depends on how the model is trained. If a particular model was mostly trained for producing software code, it can function as an AI-driven coding assistant for use in software development. For example, if it is given an input from a developer stating “Complete the function body”, then the output will contain code for the function body as generated by the model.

Numerous coding assistants, powered by LLMs, are available for use on the Internet. Two common assistants are ChatGPT [3] and GitHub Copilot [4]. Both of these assistants can be used to generate software code. However, as mentioned before, each one of these tools has its peculiarities and situations in which it shines. For example, their primary way of access differs from each other. When it comes to GitHub Copilot, it is mainly intended for use in an integrated development environment (IDE) [66] such as Visual Studio Code [67]. This is in contrast to ChatGPT, which is primarily accessed through its standalone web-based interface [3].

2.4.1 Typical Usage

Although these AI-driven coding assistants are advertised by their creators as multifaceted, it does not necessarily entail that users would consider these assistants for all sorts of purposes. Sergeyuk et al. [8] discovered that software developers would implement new features themselves, rather than instruct an AI tool to do so, as they found such a task to be enjoyable in some regard. The study also found that software developers would rather use these assistants to overcome tedious tasks like writing tests or code documentation. This shows that, while some tasks may be delegated to an assistant, software developers still prefer to perform certain types of tasks by themselves.

Davila et al. [2] conducted a survey to discover how industry practitioners perceive AI-driven coding assistants. Firstly, the responses showed that many practitioners found these assistants particularly useful for avoiding the use of search engines while coding. For example, a software developer may search the web for the usage of a specific API. Most importantly, the survey showed that respondents' biggest issues while using these assistants were largely attributed to the LLM's inability to appropriately recognize the context and disregard conformance to software requirements.

Evidently, the use of coding assistants comes with its own pitfalls, which affect their viability in practice. However, such issues can be mitigated through prompt engineering. Marvin et al. [68] describe prompt engineering as a practice that involves the use of certain techniques for improving prompts sent to an LLM. The hope is that using these techniques will result in outputs that are more appropriate and of a higher quality than the baseline outputs.

3

Related Work

3.1 Automatic Repayment of Technical Debt

OBrien et al. [6] manipulated prompts to Copilot that instructed it to solve instances of SATD expressed in the form of TODO comments. Through some very basic edits of the comments, they were able to make Copilot more effective in solving SATD. This thesis aims to use the same TODO comments from their dataset [10], repaid or otherwise, to achieve the same end goal but with code that closely aligns with the intended functionality.

Furthermore, Copilot has undergone significant changes since the aforementioned authors conducted their experiment. In particular, GitHub [69] recently announced that it swapped out its default model “Codex” with the new “GPT-4o” model for code generation tasks in GitHub Copilot. As such, the prompt engineering techniques proposed by the authors, that aimed to increase the number of successful SATD repayments, may very well be obsolete by now.

More studies have been conducted on automatic repayment of SATD by using other LLMs than the ones offered by GitHub Copilot. Sheikhaei et al. [70] fine-tuned smaller models and used prompt engineering with LLMs for automatically repaying instances of SATD. The results of their experiments, used to evaluate their accuracy, showed that the fine-tuned models struggled to keep up with the LLMs that used prompt templates. Towards the end, they suggest that the future work in this area should involve fine-tuning even larger models for the same purpose [70, Ch. 8]. In contrast, this thesis aims to create and test yet another prompt-based approach for the repayment of SATD using an LLM.

As for solving specific types of TD, Nunes et al. [71] evaluated the capability of multiple LLMs when instructed to eliminate maintainability debt inside OSS codebases. To find instances of maintainability debt, they used SonarQube [72] to check for spots in the codebase where the tool deemed there to be a maintainability-related issue. Roughly 69% of the graders, who were students enrolled in a Computer Science program, agreed that the generated code was more readable. This thesis is different because it does not restrict itself to solving certain types of TD using an LLM. Instead, it considers all sorts of TD in the same way that they are reported by OSS contributors in TODO comments.

3.2 Requirements in Open-Source Software

Xiao et al. [73] interviewed participants of a certain OSS project and collected data related to it (e.g., issues, pull requests) to create a framework for RE in OSS. Among other things, their framework showcased how requirements flow between three different practices in RE: elicitation, specification, and validation. The issue with their approach is that they have based this framework on data gathered from a single OSS project, which might have its peculiarities that are not found elsewhere in OSSD. Therefore, as stated by the authors themselves [73, Sec. 5.3], certain parts of the framework might not be generalizable to other OSS projects. This thesis considers a larger number of OSS projects, without direct interaction with contributors, for performing practices within RE.

On the topic of using NLP for finding and classifying requirements in OSSD, Vlas and Robinson [74] used two variations of their RCNL framework for this purpose. The dataset they used to evaluate these variants against consisted of documents mined from OSS projects hosted on SourceForge [75]. They found the results of their experiments to be satisfactory, however they noted that future work should focus on processing structured data using this framework [74, p. 35]. This paper shares the same end goal as this thesis: the automatic elicitation of requirements in OSSD using NLP. However, it will be accomplished differently. The difference with this thesis is that it will propose its own textual patterns for eliciting requirements, and test these patterns on the type of documents commonly found inside GitHub repositories (e.g., issues, pull requests).

4

Methods

The methodology for answering the RQs in Section 1.2 consists of three different steps outlined in this chapter. Figure 4.1 shows an overview of these steps. Firstly, Section 4.1 shows how the original dataset of TODO comments, which was published by OBrien et al. [10], is to be preprocessed. In particular, this step involves filtering the list of TODO comments based on their repositories (described in Section 4.1.1), and then downloading textual documents from their repositories (described in Section 4.1.2). After doing so, requirements are elicited from these documents by using NLP, as described in Section 4.2. The results from the execution of this particular step is used for answering **RQ 1**. Finally, prompts are generated in accordance to the process described in Section 4.3. The purpose of generating these prompts is to answer the remaining RQs: **RQ 2**, **RQ 3** and **RQ 4**.

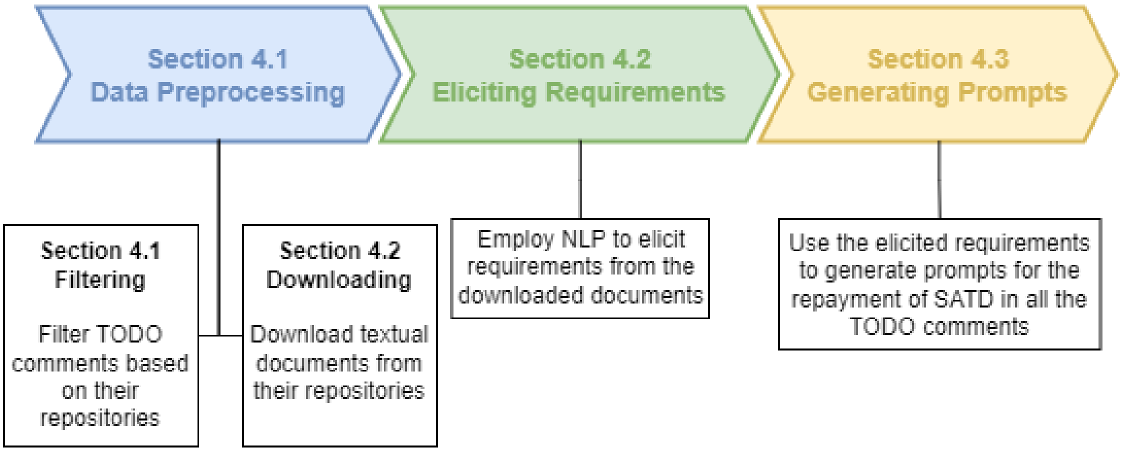

Figure 4.1: Methodology used for answering the research questions in this thesis

4.1 Data Preprocessing

The dataset used in the experiments is based on a list of 229 GitHub repositories, each containing at least one of the total 380 TODO comments. It was preprocessed by filtering the TODO comments (described in Section 4.1.1), and downloading relevant pieces of data from their repositories (described in Section 4.1.2). The entire process used practices that were either mentioned or hinted at back in Chapter 2. The scripts, used to preprocess the aforementioned dataset, were developed from scratch in Python using relevant libraries and application programming interfaces (APIs) found online.

4.1.1 Filtering

First, repositories seemingly intended for non-English speaking contributors were removed from the dataset. This was done by checking the language used in the `README` file, found inside the repository's root directory. Essentially, the file outlines important aspects surrounding the repository in question [76]. The idea was that this README indicates what (natural) language is primarily used in communications on the repository. If this file is not written in English, or if the repository does not even have this file, then the repository is excluded from further consideration in the thesis. Indeed, this approach was previously used in a paper by Wang et al. [77] in their analysis of GitHub repositories. Also, excluding repositories that did not have a single human-written document stored in an issue or a pull request made sense, especially when considering that these serve as the foundation for the elicitation process. The exclusion of such repositories early in the process ensures a minimized waste of computational resources.

It is worth mentioning that leveraging the README as an indicator of the language used in repository documents proved misleading in some cases. In particular, `PaddlePaddle/Paddle` [78] stood out as being problematic later on when downloading documents from the preliminary list of repositories in Section 4.1.2. Examples of problematic patterns noted in documents mined from this repository include issues having their title and/or body written in English, and most of the comments are written in Chinese. Therefore, the exclusion of this repository was hardcoded into the filtering script.

To accomplish the tasks described above, IDs of repaid and non-repaid TODO comments were first extracted from the `CopilotTODO.zip` archive; from `data_scripts/no_fixes.csv` and `data_scripts/yes_fixes.csv` respectively. Then, the entries in `data_scripts/2-samples.txt` were used to get the repository links to filter these accordingly. The resulting dataset of this operation reduced the amount of corresponding comments from 380 to 351; a reduction of roughly 8%.

Many repositories simply did not meet the aforementioned requirements. In some cases, they had been completely removed from GitHub. Some repositories were also renamed and/or moved to another owner, and in cases like these, the script made sure to store the new link to the repository in question.

Furthermore, some TODO comments could no longer be found in their original location and were subsequently excluded from the dataset. This reduced the number of comments from 351 to 340 (roughly −3%). There are many reasons for this reduction. Perhaps the related SATD instance was already repaid since OBrien et al. [6] conducted their experiments. Also, there is a chance that the source file was removed entirely from the repository since then.

Additionally, if the repository in question was a fork of another repository, then the link to this parent repository was stored for later use. As mentioned in Section 2.3.1, considering the parent repository and its forks when mining GitHub repositories for information is especially useful for getting a better picture of a particular software project. However, as none of the repositories in the dataset actually had a parent repository connected to them, it is reasonable to state that none of the TODO comments originated from a forked repository. Consequently, a column that stored the parent repository for a fork was not included in the output as it was empty for all rows.

While the replication package by OBrien et al. [10] was licensed permissively[1] with CC BY 4.0 [79], there was no apparent indication of how pieces of code in the repositories within this package were licensed. Section 4.1.2 involved making edits to content mined from these repositories, so the topic of repository licensing had to be addressed at this stage. The chosen approach was to manually inspect these repositories one by one and exclude any repository which were non-permissively licensed. This step reduced the number of TODO comments from 340 to 301; a reduction of about 11%.

An excerpt from the dataset, produced in this stage by the `filter.py` script (found in Section A.1), is shown in Table 4.1.

ID	**Repository**	**Filename**	**Content**
3	`CartoDB/crankshaft`	`.../glm/base.py`	TODO: untested for...
6	`aghie/tree2labels`	`baseline_utils.py`	TODO: Workaround...
7	`mkordi/pygwr`	`.../stats/runs.py`	TODO: check should...
8	`RemiAllio/Mito...`	`.../QualityIO.py`	TODO - Is an empty...
⋮	⋮	⋮	⋮
999	`gdraheim/docker...`	`testsuite.py`	TODO# too long?
1001	`ganeti/ganeti`	`.../network.py`	TODO: reserve/release...
1006	`megvii-model/MOTR`	`models/motr.py`	TODO: Make this...
1013	`.../cuckoo-modified`	`.../PDFFilters.py`	TODO: TIFF and...

Table 4.1: Excerpt from the dataset stored in `filtered.csv` (301 rows)

Figure 4.2 shows an overview of the filtering process described in this section.

---

[1] Licenses were deemed to be permissive if they allowed the distribution, modification and private use of the licensed content.

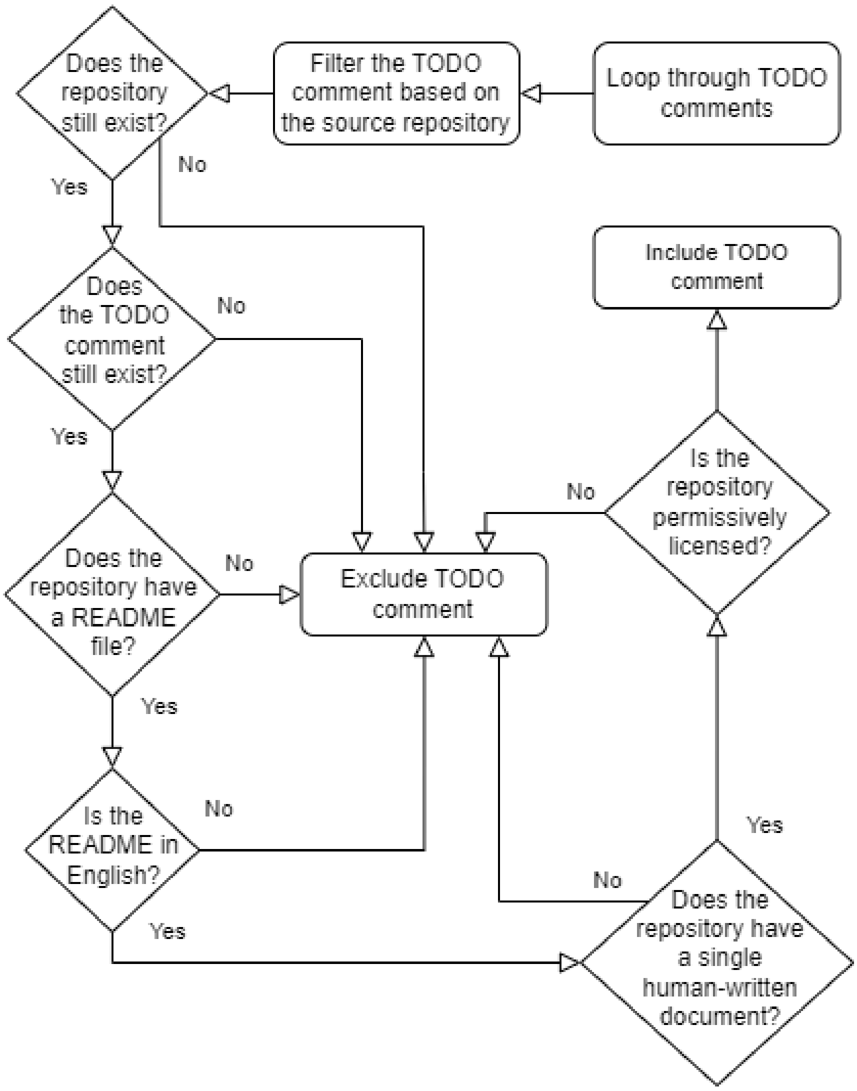

Figure 4.2: Overview of how the TODO comments were filtered

4.1.2 Downloading

Repositories found in `filtered.csv` were finally downloaded using the `download_repos.py` script (found in Section A.1). The responsibility of this script was to facilitate the collection of simple metadata surrounding the repository (e.g., license file, current HEAD commit hash), a copy of its codebase, all commits made, and, most importantly, documents found in the repository and any of its forks. The last part involved probing various locations for documents seemingly composed by human contributors. These locations included the repository issue tracker and pull request reviews. Also, commits and any of their comments were collected from the parent repository.

Firstly, the author's object type in the API [80] was used to identify if a certain document originated from a bot account and, with that in mind, should be excluded from consideration. However, commits were exempt from this rule (but not their comments). If all other types of documents breaking these rules were kept at the end of this process, then this would potentially complicate the elicitation process and cause a lot of irrelevant information to be considered.

Furthermore, certain comments found in the issues had code blocks within them. These code blocks can contain a large block of text within the comment, such as program logs (or simply software code), to assist in debugging a particular issue. Concerning this, trying to extract requirements from these code blocks (through the use of NLP tools) is essentially a lost cause.

Considering the above, the script excluded documents containing only one or more code blocks. An additional version was stored for documents that contained both code blocks and text, in which any code blocks were removed. Code blocks were detected in the comment using a regular expression. This regular expression covered the removal of both indented and fenced code blocks, as described in the GitHub Flavored Markdown Specification [81, Sec. 4.4–4.5]. Figure 4.3 shows a side-by-side example of the original version and the one without any code blocks (referred to as the `no_code` version).

~~~
These numbers both need units - if it's using LAMMPs' `real` units system (I think that's the default?), it should read
```python
    return {'potential_energy': pe * u.kcalpermol,
            'forces': forceArray * u.kcalpermol/u.angstrom}
```
~~~

(a) Original version [82]

```
These numbers both need units - if it's using LAMMPs' `real` units system (I think that's the default?), it should read

```

(b) `no_code` version

Figure 4.3: Two previews of a comment, with and without any code blocks

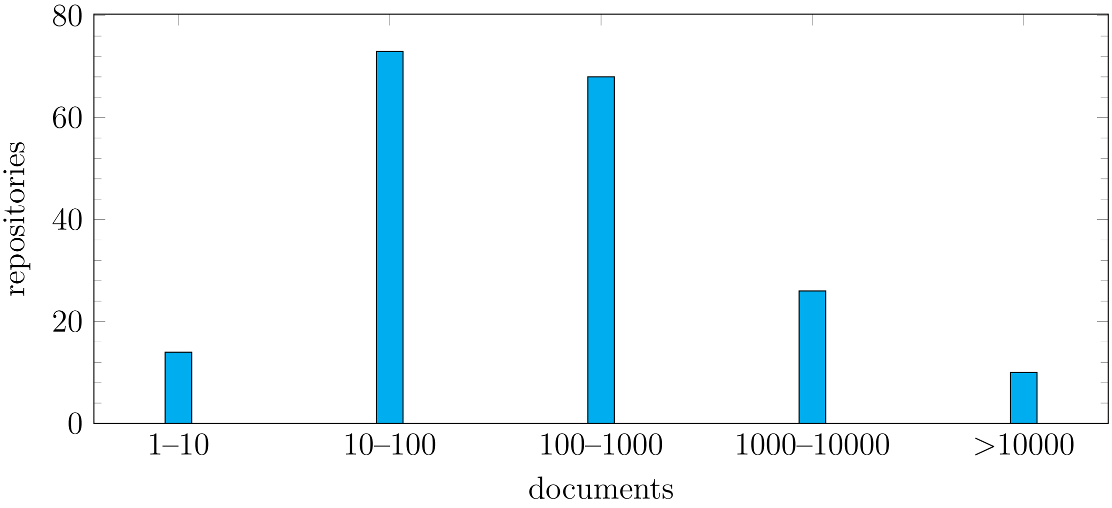

Figure 4.4: Document counts for 191 of the preprocessed repositories

The criteria for the inclusion and exclusion of different documents were as follows[2]:

1. If a bot created an *issue*, and this issue had no human comments, then the issue in its entirety (issue body and comments) was excluded. If it was created by a bot and had human comments, then the issue was fully included. Additionally, issues written by humans without any comments were included, as the issue body can still be useful for eliciting requirements. With that in mind, for all issues included, the issue body itself was considered a comment.

2. If a *commit* was created by any contributor, regardless if it was by a human or a bot, then it was included[3]. Including all sorts of commits is essential for establishing the contribution history. Comments made on these commits served as documents, and they were only included in the dataset if a human contributor wrote them.

After assembling the dataset per the aforementioned criteria, repositories without any documents whatsoever were removed using the `finalize.py` script (found in Section A.1). This reduced the number of repositories from a total of 199 to 191, and the number of TODO comments from 301 to 286. Figure 4.4 shows the ranges of document counts for all repositories. Most repositories contained 10 to 1000 documents, whereas some had over 10000 documents.

4.2 Eliciting Requirements

While it was certainly possible to manually go through every document mined in Section 4.1.2, in a pursuit to find formal/informal requirements specified within these, it would not have been optimal. Firstly, some of the mined documents found

---

[2] Same criteria for pull requests, as for issues.

[3] Commit messages only served as titles for comments sourced from those commits.

in various locations (such as the repository issue tracker) might not even intend to specify requirements, or hint towards the presence of these, in the first place. Secondly, this process introduces a problem: What kind of factors “qualify” pieces of text within a document to denote a requirement? It could be the context in which the document is found (e.g., a merged pull request) or the type of words used in the document itself (e.g., modal verbs).

There are a lot of things that can be used as a qualifying factor for the identification of requirements. As there were many documents that could all contain qualified requirements, evidently, executing this process manually would have been a waste of efforts and/or a source of misconceptions caused by the human factor. This aspect would not only be detrimental to this thesis, but also to real-world projects in OSSD. With all of this in mind, it made sense to create a semi-automated way of eliciting requirements found inside of repository documents.

Requirement documents were identified using Stanza, a toolkit published by Qi et al. [83] that can be used for running NLP on text documents. The process used in this thesis was similar to those employed by Vlas and Robinson [74, Tab. 1] in their framework for finding requirements. The first part of identifying requirements in documents involved “tokenization”, which is meant to separate pieces of text (e.g., words, question marks) in the full text to produce a set of “tokens” [84]. This paved the road for the second step, which was about looping through these tokens and seeing which indicated the presence of a requirement within the document through two heuristics:

1. The first condition checked if the token was a modal verb (e.g., “should”). Possible modal verbs are defined per The Penn Treebank, published by Marcus et al. [85]. In essence, such words aim to describe things such as a necessity [86].

2. The second condition uses a list of more qualifying verbs that were not included in the Penn Treebank (e.g., “want”), which also could indicate the presence of a requirement.

Only documents that fulfill either one of the two conditions above were included in the script’s output. The relation between multiple documents found in the same source was not taken into account for their mutual inclusion. For example, a comment on a particular issue might fulfill one of the conditions and end up included in the list of identified requirements. However, the issue body itself is not included unless it also manages to fulfill either condition.

The presence of a requirement within a document was not enough to justify its inclusion in the resulting output by the script. The document also had to be, in some way, relevant to the TODO comment in question. This was done by calculating the relevance value, which is set to be equal for all requirement documents within a single document group (e.g., a specific pull request and its comments). Relevance was evaluated as high (`relevance=1.0`) if one of the commits associated with the requirements document had introduced the TODO comment, and medium (`relevance=0.5`) if they affected the file in which the TODO comment was found. Additionally, relevance was low (`relevance=0`) if neither of these conditions was fulfilled. If the relevance was evaluated as low, the document was excluded.

Calculating the relevance value was straightforward for commits. On the other hand, pull requests can have multiple commits associated with them, but issues cannot be directly connected to commits. This was solved for pull requests by looping through its commits and using the highest relevance value found in a particular commit. As for issues, these could reference relevant PRs and/or commits in its body or any of the comments. Therefore, any issues with a body and/or comments doing so inherited the maximum relevance value from all the referenced documents.

For each commit, the creation date of the commit was stored together with its relevance value. As pull requests and issues obtained their relevance values from these commits, the date of their oldest (referenced) commit was also stored in the resulting requirement document. This is because users on GitHub have reported that the platform fails to properly handle commits containing file renames [87]. Instead of marking these files as being renamed, it marked files as being added/deleted. If the script did not store the dates of relevant commits and preferred documents that referenced older commits, then there was a chance that one would get further away from the “true source” of the TODO comment. This had to be done in preparation for when these documents were selected in Section 4.3. Otherwise, there would have been a risk of selecting documents that renamed a particular TODO file over documents that had actually introduced the TODO comment in the first place.

Finally, it made sense to consider the importance of the requirements document, as not all authors of a certain document share the same degree of “power” (e.g., push access). This served as a way of conducting stakeholder analysis. The importance was calculated for every document, according to the enum value of their `author_association` field [88]:

1. Documents, written by owners, had the highest importance value (`importance=1`).
2. Those by collaborators (users with read/write access to the repository) had a medium-high value (`importance=0.67`).
3. Contributors, which are users who had previously had a commit merged into the repository, had a medium-low value (`importance=0.33`).
4. Any other authors, such as those who never contributed to the repository before, had the lowest importance value (`importance=0`).

Figure 4.5 shows a simplified overview of the elicitation process employed in the `elicit_reqs.py` script (found in Section A.1).

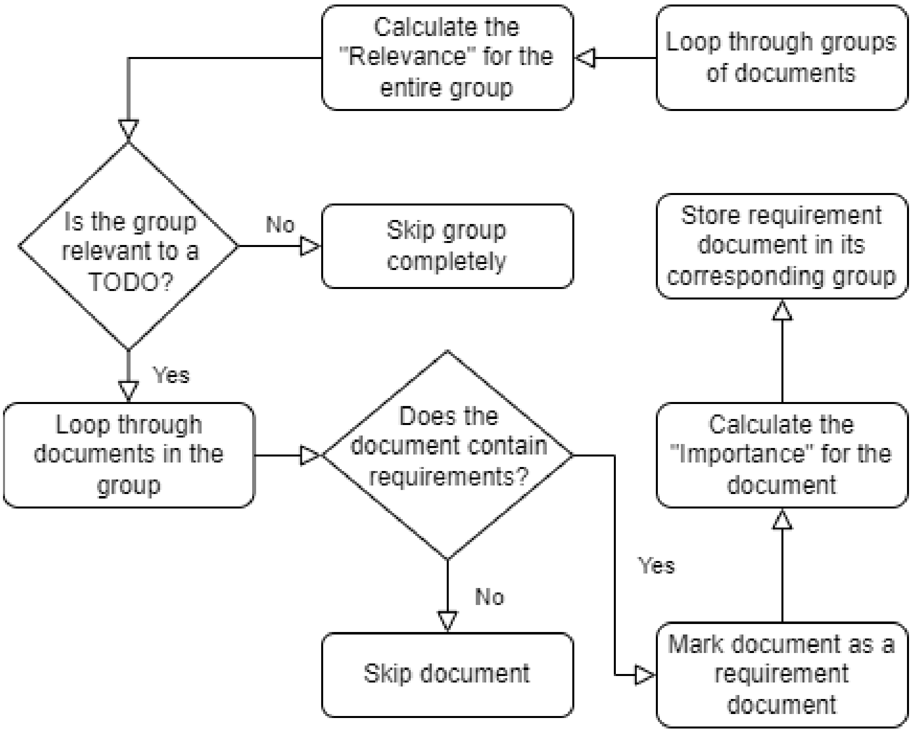

Figure 4.5: Overview of the automated requirements elicitation process

4.3 Generating Prompts

It was impossible to include all the requirement document groups in the prompts to Copilot when answering **RQ 3** and **RQ 4**, as some TODO comments had a bunch of requirements associated with them. In particular, **RQ 3** only used one group to resolve the SATD associated with a particular TODO comment. And, as for **RQ 4**, this used three groups to see if there are any differences compared to simply using one group. Therefore, both of these questions required a way to select the most optimal subset of document groups by using the variables at hand. This was made possible using the `select_reqs.py` script (found in Section A.1).

Groups of requirement documents were sorted according to their relevance in descending order. This ensured that the selection process prioritizes documents that are relevant over those that are not. Then, the groups were finally sorted after the date on which their referenced commits were created, but now in ascending order. Groups were also sorted according to the cumulative importance of all documents within them (in descending order). Then, random values sort the groups, so there was no risk of only getting requirements sourced from the same location (e.g., commits) at the top of the list.

Finally, prompts were generated for the TODO comments pertaining to the elicited requirements. This was done by creating prompt templates for answering **RQ 2**, **RQ 3**, and **RQ 4**. These are shown in Figure 4.6. All prompt templates had four fields in total. The last two fields of the prompt template, "Resolved" and "Comments", were the most important ones when answering the RQs:

1. The “Resolved” field describes whether the SATD has been repaid in a way that conforms to the requirements in the first three groups. The resolution may be partial, i.e., the output from Copilot solved the SATD instance but not in a way that conforms to the requirements. Regardless of whether Copilot managed to output code that conformed to the requirements, the prompt was deemed not to have solved the issue at hand if the SATD was not actually repaid. In reality, some successful resolutions may not have truly been successful. In particular, none of the generated code was executed. Even if there was a nearby code comment specifying that the approach used in the generated code might not actually work, the resolution might still be marked as successful.

2. The “Comments” field was used to add thoughts on why the resolution was either a success, a partial success, or a failure. This made it possible to conduct a qualitative analysis alongside the quantitative one. For example, it was possible to reason about potential shortcomings of the elicitation process and how it may have caused Copilot to output code that did not solve some SATD.

These templates were partially filled out using the `gen_prompts.py` script (found in Section A.1), and manually sent as prompts to the GitHub Copilot extension in Visual Studio Code [89] (as done in the experiments by OBrien et al. [6]). After doing so, with Copilot set to run in “Edit”-mode with the GPT-4o model, the remaining fields of the prompt template were filled out.

```
-----Input-----
There is a TODO comment stating '<todo_content>' in the current script
    '<script_name>'.
Provide code which solves the issue that is described by this TODO
    comment.
-----Output-----

-----Resolved (Yes/Partially/No)-----

-----Comments-----
```

(a) First prompt template

```
-----Input-----
There is a TODO comment stating '<todo_content>' in the current script
    '<script_name>'.

Below is one group of requirement documents related to this comment:
<first_req_group>

Provide code which solves the issue that is described by this TODO
    comment. Use the specified group of requirement documents as a
    source of context.
-----Output-----

-----Resolved (Yes/Partially/No)-----

-----Comments-----
```

(b) Second prompt template

```
-----Input-----
There is a TODO comment stating '<todo_content>' in the current script
    '<script_name>'.

Below is three groups of requirement documents related to this comment:
This is the first group:
<first_req_group>
This is the second group:
<second_req_group>
This is the third group:
<third_req_group>

Provide code which solves the issue that is described by this TODO
    comment. Use the specified groups of requirement documents as a
    source of context.
-----Output-----

-----Resolved (Yes/Partially/No)-----

-----Comments-----
```

(c) Third prompt template

Figure 4.6: The three prompt templates used to generate prompts

5

Results

5.1 Automatic Elicitation of Requirements

After eliciting groups of requirement documents for every TODO comment, those without at least three groups were excluded from the dataset. In total, 190 TODO comments were excluded, which meant that 96 TODO comments remained. The number of unique repositories, in which these TODO comments could be found, was reduced from 191 to 71 (a reduction of 63%).

The number of document groups that were found to be in some way relevant to each TODO comment is shown in Figure 5.1. Overall, it was surprising that some of these TODO comments had over 100 relevant groups. It was also essential to consider the actual number of requirement documents stored in all three groups that were selected for every TODO comment. Figure 5.2 shows that no TODO comment had over 100 documents. In fact, the vast majority of these TODO comments actually had between 3 and 25 relevant requirement documents.

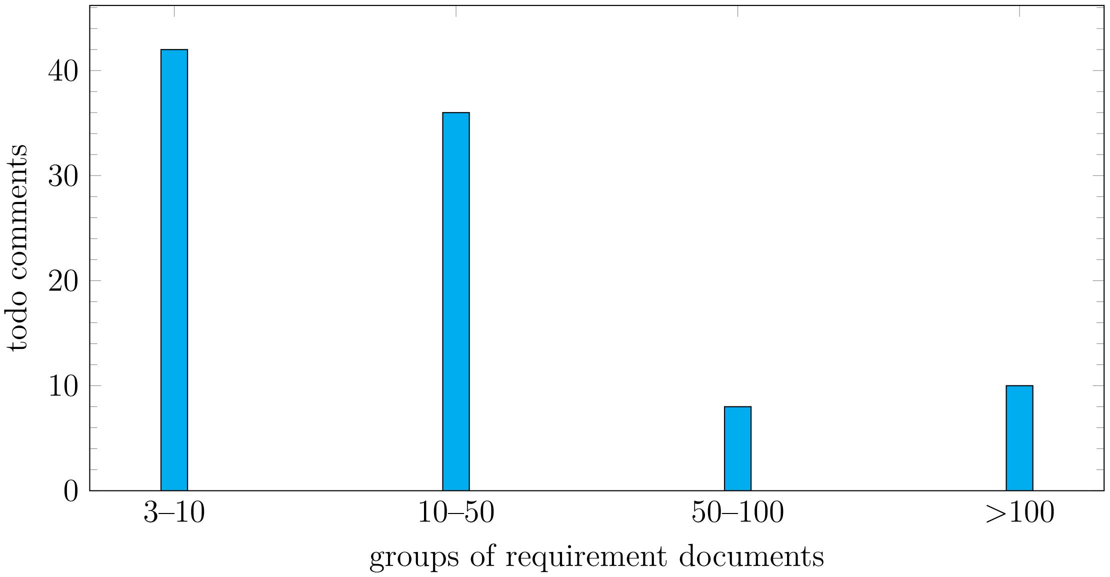

Figure 5.1: Groups of requirement documents for the 96 TODO comments

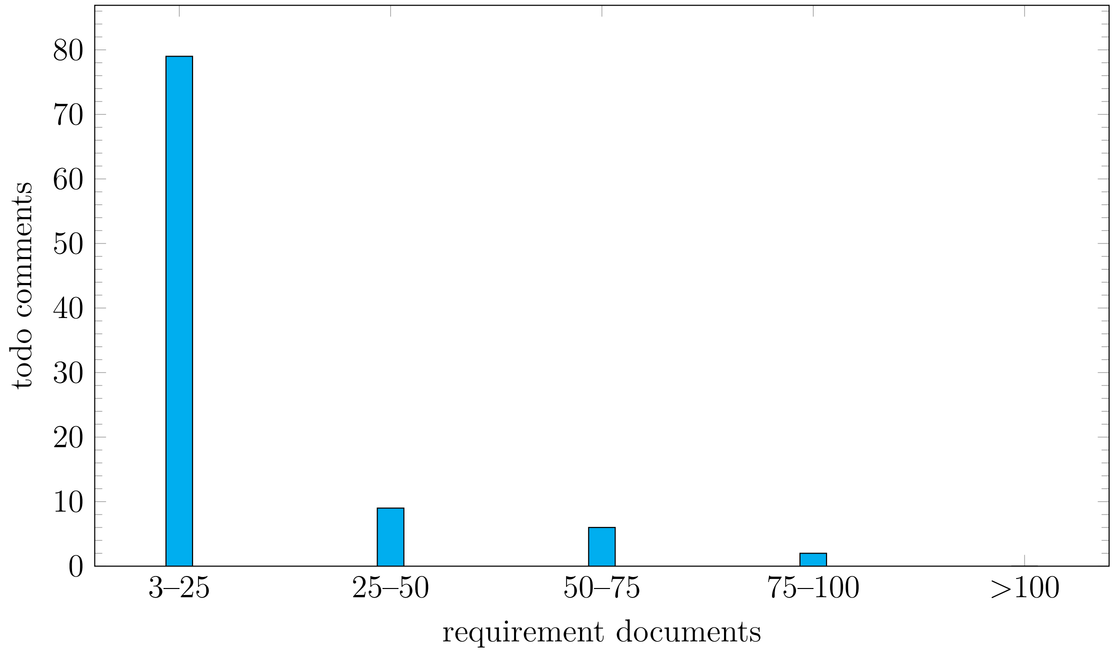

Figure 5.2: Number of requirement documents for every TODO comment

Interestingly enough, Figure 5.3 shows that most groups of requirement documents were sourced from pull requests.

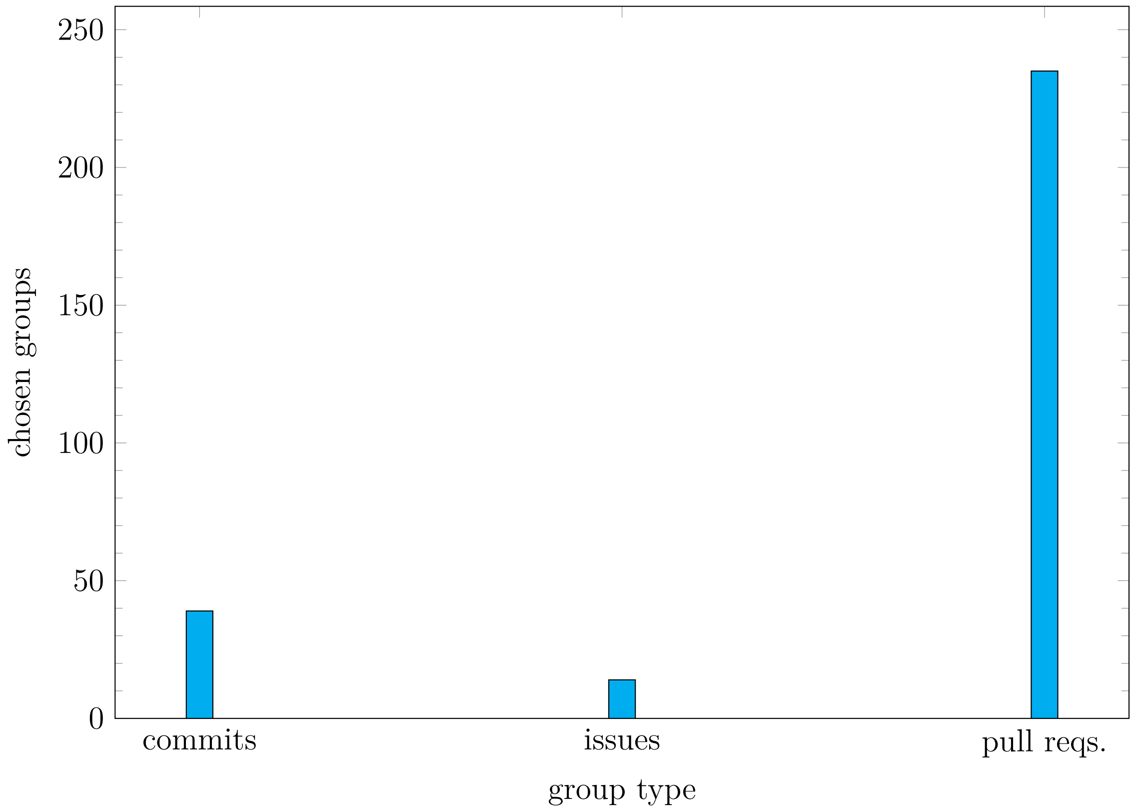

Figure 5.3: Counts for the types of the chosen groups of requirement documents

Figure 5.4 shows three groups of requirement documents stored for the TODO comment with ID 610.

```
Title: 'Exp Factory WIP :construction: establishing filter pipeline'. Group type: 'Commit'. Relevance: 'Most relevant'.
Requirements:
Type: 'Comment'. Importance: 'Important'. Content:
never mind, I saw that the type changed recently. I'll fix it in `exp_factory`
```

(a) First requirement group

```
Title: 'Feature/ipp channelizer'. Group type: 'Pull request'. Relevance: 'Most relevant'.
Requirements:
Type: 'Body'. Importance: 'Somewhat important'. Content:
Should work on Windows and Linux for now. @dieris please check with real data.

Type: 'Comment'. Importance: 'Important'. Content:
As far as I can see, this is working fine with real data. A comparison of readout fidelities is still pending
```

(b) Second requirement group

```
Title: 'Holding on writer file in cals'. Group type: 'Issue'. Relevance: 'Most relevant'.
Requirements:
Type: 'Comment'. Importance: 'Somewhat important'. Content:
Seems to be caused by h5py/h5py#775. Will be fixed when h5py 2.7.0 is released.

Type: 'Comment'. Importance: 'Important'. Content:
Replacing data writers with data buffers as @grahamrow suggested may be enough for now. I'd prefer not to save calibration data anyway
```

(c) Third requirement group

Figure 5.4: Examples of three requirement groups for a randomly-selected TODO comment

5.2 Repayment of SATD with Copilot

After prompting Copilot by using three different prompt templates for each of the TODO comments, statistics pertaining to the resolutions of these comments were collected with the `prompt_stats.py` script (found in Section A.1). Figure 5.5 shows the output of this script.

Outputs from the first prompt template, which did not use any requirements, had the most successful resolutions. However, it also had the most significant number of partial successes, meaning it was the least effective template for solving SATD in a way that conforms to the selected requirements. In contrast, the third prompt template had the fewest partial resolutions. While it had almost the same number of successful resolutions as the first prompt template, the third template recorded more failed resolutions.

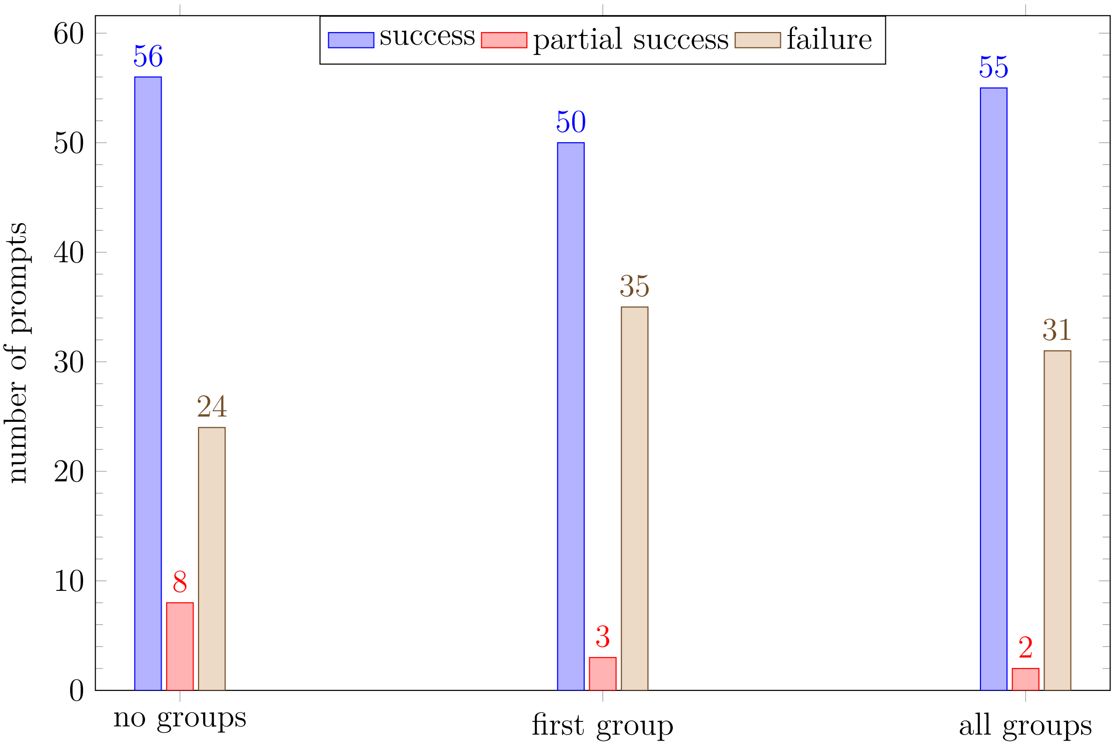

Figure 5.5: Resolutions for each of the three different prompt templates

5.2.1 Prompts and Comments

This section presents examples for prompts generated from each of the three prompt templates, along with comments that were recorded for many of the resolutions. For all 264 prompts, there was a total of 141 comments. Presenting the most notable comments was vital for the qualitative analysis performed later in Chapter 6. In cases where there was nothing significant to comment on for a particular prompt, the comment field was left empty. Figure 5.6 shows three Copilot outputs for all the three prompts for the TODO comment with ID 722.

~~~
I will modify the code to append an empty texture for the slot when a texture is missing in the `nodeNames` dictionary.
````
<bunch of code>
````

Made changes.

The TODO comment has been addressed by appending an empty texture for any missing slot in the `nodeNames` dictionary. If you need further assistance or additional modifications, let me know!
~~~

(a) Output for prompt from the first template

~~~
````
<bunch of code>
````

Made changes.

The TODO comment has been addressed by appending an empty texture for missing PBR node slots to ensure compatibility. Let me know if further adjustments are needed!
~~~

(b) Output for prompt from the second template

~~~
I will modify the code to append an empty texture for the slot when a texture is missing, ensuring compatibility with PBR nodes and addressing the issue described in the TODO comment.
````
<bunch of code>
````

Made changes.

The code has been updated to append an empty texture for missing slots, ensuring compatibility with PBR nodes as described in the TODO comment. Let me know if further adjustments are needed!
~~~

(c) Output for prompt from the third template

Figure 5.6: Outputs by Copilot for all the three prompts, taken from a randomly-selected TODO comment

5.2.1.1 No Groups

This template, which did not have any groups of requirement documents included in its prompts, accounted for a total of 24 failed resolutions across all the TODO comments. While it had the fewest failed resolutions of the three templates, it was relevant to consider the potential reasons for these failures. Some failed resolutions were seemingly prevalent in instances where the script, in which the TODO comment was located, had many (usually between 1000 and 4000) lines of code. In cases like these, Copilot was often found to start working in completely unrelated code blocks.

Moreover, if the TODO comment was ambiguous, Copilot would make heaps of changes that ended up breaking the script's functionality. If it did not do that, it would output code that was not placed in the correct location. Multi-line TODO comments, which were not captured correctly in the original dataset, happened to be the cause in one instance where Copilot broke the functionality.

As for the 8 partial resolutions, some of these were found to be caused by generated code that did not conform to certain NFRs. In one instance, Copilot failed to generate usage examples in the code comments of a specific function, per a requirement specified in the third group. In another instance, the first group of requirements had specified a need to make code maintainable and readable. The original version of the script, in which a specific TODO comment was located, already failed to fulfill these two aspects. However, after Copilot's edits, the script became much worse regarding its maintainability and readability.

Furthermore, Copilot had failed to format some code per the PEP 8 style guide [90] in another instance. This was specified in the first group of requirement documents. The remaining five partial resolutions were caused by generated code that did not conform to certain FRs, such as retaining important arguments to avoid breaking the original functionality.

Some successful resolutions had comments that are also worth mentioning. In contrast to what was mentioned before, Copilot actually had no difficulties tackling larger scripts. Inserting the generated code in the correct position was seemingly not an issue at all. Furthermore, Copilot produced code that conformed to requirements (both FRs and NFRs) that it had no access to in the first place.

However, there were uncertainties about whether the resolution was truly successful. In one instance, the generated code was written in a specific way that the repository contributors had suspected would not work. In another instance, after manually reviewing the elicited requirements for another TODO comment, the requirements were largely irrelevant to the TODO comment in question.

5.2.1.2 First Group

The second prompt template included a single group of requirement documents inside its prompts. This template resulted in 35 failed resolutions in total, the highest failure count recorded across all the templates. Most of these failed resolutions were seemingly caused by Copilot's inability to focus on solving the SATD instance while following the requirements. Copilot was also noted to be much more "confused" when given prompts generated using this template.

For example, it would start working on completely irrelevant pieces of code. Sometimes, Copilot managed to generate the correct code that would solve the SATD but ended up inserting it in the wrong part of the script. Moreover, some of these failures were caused by the fact that requirements in the first group were largely irrelevant to the TODO comment at hand.

Furthermore, this template had three partial resolutions, less than half the number recorded for the first prompt template. The first instance was caused by Copilot's inability to generate examples in the documentation for a function, as specified in an NFR in the third group of requirement documents. The partial resolution in the last instance was caused by the fact that the generated code actively worked against the requirements specified in the first group.

As for the second instance, this was seemingly also caused by the lack of access to certain requirements in one of the groups. However, it was noted that Copilot could have captured this requirement indirectly by inspecting the script. In particular, a requirement in the second group had specified that errors should be logged appropriately (which is an FR) – Copilot did not see a need to do so in the generated code, even though this was done elsewhere in the script.

Many of the comments on the successful resolutions mention Copilot's ability to generate code that follows NFRs specified in the first group of requirements, such as readability and maintainability. It also seems to focus on these NFRs, even if the prompt does not contain any requirements that tell it to. This was presumably caused by including the word "requirement" in the prompts. Moreover, Copilot sometimes refused to modify the code if it was phrased as a question and decided to make a small comment as to why it did not commit any edits. This was found to be positive, as not all instances of SATD always denote a functional problem.

Certain successful resolutions in this template suffered from issues similar to those in the first template. Firstly, the generated code in one instance had set the default value of a parameter to a value that a code comment said it could not assume. In the comment for that instance, it was speculated that these code comments could have served as an additional source of requirements. Secondly, requirements in the first group were occasionally found to be irrelevant to the TODO comment at hand. This made it hard to determine whether the resolution was successful or only partially successful.

5.2.1.3 All Groups

The final prompt template accounted for 31 failed resolutions in total, placing it closer to the second template and further away from the first. Some of these failures were caused by Copilot inserting the code, which would otherwise solve the SATD instance, in the wrong area of the script. That particular problem was even more pronounced if the script was large. This seemed to align with failures in the other templates.

Additionally, irrelevant requirements seemed to have caused many of these failed resolutions. In one particular instance, documents from a GitHub bot were incorrectly identified as containing one or more requirements. In other instances, the great number of irrelevant requirements in all three groups impeded Copilot's ability to properly repay the SATD, especially so if the script was also large. In another instance, in which the TODO comment contained instructions on how to solve the SATD, Copilot generated code that does the opposite of what the TODO comment suggested. This was thought to be caused by confusing requirements in the prompt.

The 2 partial resolutions were both caused by Copilot working against the requirements included in the prompt. The cause behind this behavior was not identified. However, the positive thing was that this template ended up having the fewest partial resolutions compared to the other templates.

Certain successful resolutions demonstrated Copilot's ability to identify NFRs. In cases where the requirements specifically wanted code to be properly formatted according to some styling guide (e.g., per the PEP 8 style guide [90]), Copilot did so. Also, simply including the word "requirement" would have Copilot ensure that the generated code was nicely formatted, like the second template.

However, in a few instances, Copilot's edits were not proportional. Sometimes, it added or changed a tremendous amount of code to the script, whereas prompts from the other templates produced code that did not have this issue. In one particular instance, some code was deleted from the script without any justification in the prompt response.

In one specific instance, Copilot solved the SATD but did so in a way that went against what a nearby comment suggested. This caused some uncertainty about the code and whether it actually worked as expected, which is a problem that had been observed in the other prompt templates.

5.2.2 Exclusions

A total of 8 TODO comments were excluded, as they were deemed infeasible for use. Each of these exclusions was commented on, and these comments aimed to describe the justification for excluding them from the experiments in the first place. Those reasons are:

1. Four TODO comments did not even indicate the presence of SATD. For example, one of these four comments just stated how a specific block of code works.

2. Three TODO comments could not be processed because the amount of code in the script made Copilot go haywire. Even when using prompts from the first template, which did not have any requirements included in the prompt, Copilot would still go into a loop where it repeatedly modified the (wrong) script.

3. The last TODO comment stated "TODO:", whereas there were many such

TODO comments in the script, with no indication of which ones to process. Furthermore, it was unclear which of these TODO comments actually indicated the presence of SATD.

6

Discussion

6.1 RQ 1: To what extent can three groups of requirement documents, related to TODO comments, be automatically elicited from their GitHub repositories?

After the elicitation process, only 71 of the 191 repositories were kept. A reduction of 63% seems to align with the findings of studies conducted in this particular area:

1. Firstly, a part of the reduction can likely be attributed to the fact that most GitHub repositories are used for hosting personal projects, as shown by Kalliamvakou et al. [44]. The (informal) specification of requirements may not be so prevalent in such repositories, especially if they do not have clear contribution guidelines that help enable collaboration. In particular, contribution guidelines are an aspect that Coelho and Valente [36] found to be very important for an OSS project to succeed.

2. Secondly, another part of the reduction can likely be explained with the findings by Scacchi [60] concerning requirements in OSSD and how they are decentralized. This appears to be the case here, as certain repositories had documents containing a few words and a link to a forum post. Just a few words were enough for certain repository documents to indicate the presence of a requirement. Therefore, completely refusing to pull information from external sources might lead to less context to include in the requirement documents referencing these sources.

As pointed out in Section 5.1, most groups of requirement documents were sourced from pull requests. This makes sense for two reasons. Firstly, issues are only relevant if they directly reference a relevant pull request and/or commit. Indirect references, specified in natural language, were not captured. Secondly, engaging with contributors in commits seems more complicated than trying to do the same in pull requests. This difference appears to be a product of how GitHub has built pull requests to foster collaboration in numerous ways (e.g., through pull request reviews).

To this day, there has not been a similar study that automatically elicits requirements from OSS repositories hosted on GitHub. For example, Vlas and Robinson [74] had done so only using repositories on SourceForge. As there are a wide variety of differences between different OSSD platforms, it is a positive thing that this thesis highlights a way of eliciting requirements by using the different variables specific to GitHub's platform. The approach here could help shape future work that intends to use requirements in OSSD to fulfill similar (or different) purposes.

6.2 RQ 2: To what extent does Copilot repay SATD reported by TODO comments, by generating code that conforms to requirements in all groups of requirement documents?

Prompts generated from the first prompt template accounted for the most partial resolutions. Certain partial resolutions seem to stem from Copilot incurring or worsening different types of debt (e.g., documentation debt). This suggests that Copilot is incapable of gathering and using relevant requirements by itself while repaying SATD without these requirements being explicitly specified in the prompt.

The problem described above seems to align with the survey conducted by Davila et al. [2], in which the respondents felt that Copilot failed to recognize requirements from the context. On the other hand, many successful resolutions seem to indicate that Copilot has some ability to infer relevant requirements to use when generating some code. This shows that this particular issue is not black-and-white.

When focusing solely on the repayment of SATD, with no consideration to whether the generated code conforms to the requirements, 64 of 88 instances managed to be repaid by Copilot (roughly 73%). This is a massive leap from the results of the experiments OBrien et al. [6], in which they only managed to solve SATD expressed in roughly 32% of the TODO comments. However, only a fraction of the TODO comments had been processed in this thesis. Therefore, it is hard to say whether these findings are representative for the repayment of SATD expressed in all the 380 TODO comments published by the authors [10].

6.3 RQ 3: How effective is Copilot's repayment of SATD, while conforming to requirements in all groups of requirement documents, if only the first group was included in the prompt?

The second prompt template accounted for the most significant number of failed resolutions. This was seemingly caused by Copilot's inability to focus on solving the SATD, and the requirements included in the prompt. However, it is strange that the third prompt template did not display a similar problem, especially since it uses even more requirements in its prompts. For an unknown reason, the issue of Copilot inserting code in the wrong areas of the code also appears to be more pronounced.

Moreover, prompts generated with this template had the second-least number of partial resolutions. Most partial resolutions had code that failed to conform to one or more FRs. The fact that many of the successful resolutions were related to Copilot generating code that conforms to NFRs suggests that Copilot is much more capable of following NFRs than FRs.

A possible reason for Copilot's ability to follow NFRs seemed to be the inclusion of the word "requirement" in the prompt. This pattern was also seen with prompts generated from the third template. Despite any of the differences between the Copilot model used by OBrien et al. [6], and the one used in this thesis, the process of making simpler changes to the prompt to Copilot for a better result still seems to be relevant.

6.4 RQ 4: How effective is Copilot's repayment of SATD, while conforming to requirements in all groups of requirement documents, if all these groups were included in the prompt?

Many of the failed resolutions, from prompts generated using the third prompt template, were seemingly caused by the number of irrelevant requirements in all three groups of requirement documents. The comments on these failures suggest that there are multiple reasons as to why these requirements could have been irrelevant:

1. Firstly, an attempt was made to filter away automated comments by using the GitHub API for checking whether a particular user is a bot. However, Golzadeh et al. [48] showed that this method fails to recognize bots operating through a "machine user account" [49]. After a manual inspection of the requirement documents and seeing the way that many documents from bots managed to pass through the filter, this still seems to be the case.

2. Secondly, certain groups of requirement documents were found to be outright confusing. This is probably caused by the fact that documents that were not found to contain any requirements did not end up being inserted into a group with the documents that did contain requirements. Meaning, any confusion is likely a result of removing parts of the context within these groups.

The number of partial resolutions was not much different from the previous template. Prompts from the second template had 3 partial resolutions, whereas prompts from this template accounted for 2 partial resolutions. However, the former had fewer successful resolutions. Therefore, this template seems to produce prompts that yield more repayments of SATD. This could be due to an additional amount of context added in the prompt by documents from the other two groups of requirement documents, which ends up helping Copilot in repaying SATD. Either way, including all three groups of requirement documents in the prompt to Copilot seems to have helped it the most in generating code that conforms to those requirements.

In certain successful resolutions, a peculiar issue caused Copilot to make a tremendous number of edits. This problem was not seen in the prompts from the second prompt template. It could be that including three groups of requirement documents in the prompt is too much. Also, the number of requirement documents in the individual groups might play a factor in inducing this issue.

As for its ability to repay SATD, with no regard to the requirements, prompts from this template managed to do so for 57 of the 88 TODO comments (roughly 64%). This fraction is slightly higher than for the prompts in the second template, which repaid SATD in 53 comments (roughly 60%). In both cases, the fraction of SATD solved is lower than that obtained with prompts from the first template. However, prompts from the second and third templates have fewer partial resolutions. This suggests that Copilot either tries to repay SATD in accordance with the specified requirements or refuses to repay SATD full stop.

6.5 Threats to Validity

This thesis has certain threats to validity that are worth mentioning. These may explain why prompts from a particular template had more successful resolutions than those from another template. Not all the threats may be directly actionable; instead, they help denote the need for thoughtful consideration surrounding one or more aspects of the methodology used to perform the experiments.

6.5.1 Construct Validity

The premise of SATD, as defined by Potdar and Shihab [5], is that a software developer admits that there is some technical debt inside the code for a certain script. This thesis gives an example of one way to detect SATD, that is, through TODO comments. However, it could be the case that the developer does not actually believe that this TODO comment denotes some technical debt. At the time of writing this thesis, there is a lack of research focusing on the validation of SATD. This would have to be done, in order to properly assess the contributions of experiments conducted on instances of SATD.

6.5.2 Internal Validity

Firstly, the elicited requirements had not been validated. This could have been done through stakeholder interviews, i.e., asking the authors of all documents if the requirements within truly reflect their expectations of how the software should function. As this task would require a significant amount of time to complete, or be outright impossible (e.g., stakeholders are not willing to participate in interviews), requirements validation was not considered for this thesis. The result is that the elicited requirements might not reflect the intended functionality (or only a part of it), which is precisely what **RQ 1** aimed to capture in the first place.

As mentioned in Section 4.2, modal verbs from the Penn Treebank [85], and a list of additional qualifying verbs, were used for detecting the presence of requirements in documents. It is possible that using some of these qualifiers resulted in documents wrongly being marked as a requirement document. Perhaps certain verbs that could have served as qualifiers were overlooked, and this caused some requirement documents to not be marked as such.

Some successful resolutions were noted to have code comments nearby hinting that the way in which Copilot repaid the SATD would not end up working. As requirements were not also sourced from code comments when answering **RQ 1**, it did not make sense to label those instances as being partially resolved. Considering the resolutions to be complete failures was out of the question too, because only the output from Copilot (and where it was inserted) were used for evaluating the repayment of SATD. The result of not eliciting requirements from code comments is that all the intended functionality may not have been captured to the greatest extent possible.

Furthermore, the process of filtering documents is far from perfect. In particular, certain documents stemming from issue bodies did not actually contain any requirements, and were seemingly only recognized as such because of words used in the issue template chosen by the author [91]. As shown through the surveys conducted by Li et al. [92], contributors often perceive the task of filling out issue templates as a chore. This could serve as a reasonable explanation as to why certain requirement documents contained issue templates with only some (or none) of the fields filled out by the author. Therefore, when mining different types of documents from GitHub, there is an apparent need to develop and use a more sophisticated filtering method; covering more than just the removal of code blocks from documents. There is also a possibility that removing code blocks from documents may have resulted in certain requirements being incomplete.

As discussed in Section 6.4, the detection (and subsequent removal) of documents authored by GitHub bots was deemed to be inaccurate. The tooling for doing so, available when writing the methodology in Chapter 4, did not show promising results when testing it against certain comments known to be made by “machine user accounts” [49]. Therefore, the human/bot classification for authors of comments was ultimately up to the GitHub API. It is possible that this issue could have been solved by creating a specialized tool that acted on the comments made by a particular user. For example, if a user is found to make hundreds of similar comments in a repository, then this could indicate that the user is, in fact, a bot.

Moreover, it could be the case that the algorithms used for calculating the importance and relevance of requirement documents were flawed. This is not something that was found to be apparent while prompting. However, it is possible that a more sophisticated calculation of those variables would have yielded more appropriate groups of requirement documents to use. This could particularly be the case when considering that those algorithms had not been tested outside the experiments conducted within this thesis.

The repayment of SATD and/or conformity to requirements may have been assessed in a biased manner. Considering that SATD was repaid in 73% of the TODO comments, whereas OBrien et al. [6] only repaid SATD in 32% of the comments they processed, this could be the case. However, considering that only 88 of the total 380 TODO comments were processed in the experiments, this might not be a fair comparison. Also, such a significant difference is most likely to be caused by improvements between the two different models used. In particular, the authors stated that they used Copilot with the OpenAI Codex model, based on GPT-3 [93]. The newer GPT-4o model [94], which was used in this thesis, dubbed “Codex V2”, suggests that significant improvements have been made to improve its capabilities. Even if these factors were accounted for through the use of Cohen’s kappa and multiple assessors, LLMs are inconsistent by their very own nature. As such, an unbiased evaluation might not mean much if the code generated by Copilot differs greatly in-between different sessions.

Finally, it is possible that the prompt templates would have yielded different results if they made use of different numbers of groups. In particular, the first prompt template (used for **RQ 2**) served as the control group, whereas the second prompt template (used for **RQ 3**) and third prompt template (used for **RQ 4**) served as treatment groups. The control group contained no groups of requirement documents, however, the treatment groups contained one and three groups respectively. Perhaps the most optimal treatment, yielding the greatest number of successful resolutions, could have been achieved through a prompt template that contained two groups of requirement documents (or maybe even four of them). The formatting that the templates followed could have also played a role in affecting the resulting resolutions.

6.5.3 External Validity

Firstly, certain requirement documents were incomplete beyond any doubt. This was indirectly caused by choosing not to consider external sources for the elicitation of requirements. As such, this particular restriction makes the findings of this thesis not only inapplicable for repositories with a plethora of documents that contain references to external sources, but also for entire repositories that are directly dependent on external sources (e.g., external issue tracker). In fact, Kalliamvakou et al. [95] and Käfer et al. [96] discovered that communication about the development of software projects hosted on GitHub, for the most part, actually happens outside of GitHub itself.

There are ways that could potentially help alleviate issues stemming from the consideration of external sources within documents. In the case of sources with restricted read access, which could possibly be detected through the response by checking if the status code was “401 Unauthorized” [97], one may decide to completely exclude the document in question from the group. As for external sources that return a “404

Not Found” response, which indicates that the requested page no longer exists [98], one can use a web archival tool such as the Wayback Machine [99]. In doing so, it becomes possible to avoid the exclusion of GitHub repositories that have their issue tracker disabled in the repository settings [100]. For example, it would be possible to consider repositories that simply serve as mirrors.

Furthermore, TODO comments located inside repositories that were not primarily meant for English contributors had been excluded from the experiments. This was done because the scripts for requirements elicitation were only written and tested for English documents. As such, the findings from this thesis may not be applicable for repositories that are mostly meant for non-English contributors. It would be interesting to see if similar experiments can be conducted for repositories in languages other than English. Certain repositories can also present themselves to be multilingual. As pointed out in Section 4.1.1, such a repository had to be excluded because most of the documents were in Chinese – with a small fraction of them being in English. Perhaps there is a way of using Stanza [83], which is the tool used for executing NLP in this thesis, in a way that supports multilingual repositories. According to the developers of Stanza [101], the tool has support for 80 languages. Moreover, the tool can be used to identify the language used in a document [102]. In doing so, one can switch the Stanza pipeline used depending on the language that was identified.

It is important to note that only a smaller subset of the TODO comments in the original dataset, published by OBrien et al. [10], had been processed in the experiments. Many of the original TODO comments were removed from the dataset for various reasons, two of which are mentioned in this section. Mitigating the problems related to these would presumably result in even more TODO comments being processed. In doing so, the results might become statistically significant for Python repositories (hosted on GitHub) that contain TODO comments.

6.6 Future Work

The intention was for the elicitation of requirements to be supported by a framework created in this thesis. In doing so, it would probably have been easier to understand the elicitation process and employ this in future research. However, due to time constraints, it was not possible to properly construct and present such a framework. As such, there is clearly a need to do so in future research. This framework could make use of the RCNL framework, presented by Vlas and Robinson [61], as an inspiration (similar to how the `elicit_reqs.py` script did, which can be found in Section A.1). Also, requirements could potentially be sourced from more locations. Function documentation can serve as another source of requirements. In particular, these can be processed by walking through the abstract syntax trees (ASTs) of Python scripts [103] found inside the codebase.

7

Conclusion

Overall, this thesis serves two purposes. Firstly, it presents a way to scrape groups of requirement documents from software repositories hosted on GitHub. In this particular thesis, this is done to elicit and use requirements related to a particular TODO comment. Secondly, this thesis showcases the need to specify relevant requirements when having Copilot solve SATD, which is expressed through a TODO comment. Without doing so, there is a risk of Copilot generating code that does not conform to the requirements. For any software project, taking this risk could result in its failure further down the road.

Due to certain external factors, such as contributors abandoning repositories, the requirements had to be elicited without any contact with the repository stakeholders. Also, as a great number of documents within these repositories could denote the presence of requirements, it was necessary to automate the elicitation of requirements. At the time of writing, seemingly no research papers present the code for such a script. Therefore, one was created for this purpose by using inspiration from similar research. Despite some of its flaws, such as falsely marking certain documents to contain requirements, it serves to pave the road for similar endeavors that call for the need of OSS requirements.

After prompting Copilot to repay instances of SATD, with and without including groups of relevant requirement documents, the results showed a couple of things. Its ability to repay SATD has seemingly improved over the past few years. In some instances, Copilot was even able to repay SATD in accordance with all the relevant requirements, without these being specified in the prompt. On the other hand, passing prompts to Copilot with all the requirements specified within, made it even better at generating code that resolved instances of SATD and conformed to the requirements. The drawback of this approach was that it caused Copilot to repay fewer instances of SATD overall.

SATD cannot truly be repaid without doing so in a way that conforms to all the relevant requirements. Therefore, future research should focus on developing and employing a robust framework for the automatic elicitation of requirements. Hopefully, the method showcased in this thesis paves the road for doing so.

Bibliography

[1] Stack Overflow. “AI | 2024 Stack Overflow Developer Survey,” Accessed: Jan. 27, 2025. [Online]. Available: `https://survey.stackoverflow.co/2024/ai/#sentiment-and-usage`.

[2] N. Davila *et al.*, “An Industry Case Study on Adoption of AI-based Programming Assistants,” in *2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, Apr. 2024, pp. 92–102. DOI: `10.1145/3639477.3643648`. Accessed: Jan. 27, 2025. [Online]. Available: `https://ieeexplore.ieee.org/document/10554736/?arnumber=10554736`.

[3] OpenAI. “Introducing ChatGPT,” Accessed: Jan. 27, 2025. [Online]. Available: `https://openai.com/index/chatgpt/`.

[4] GitHub, Inc. “What is GitHub Copilot?” Accessed: Jan. 28, 2025. [Online]. Available: `https://docs.github.com/en/copilot/about-github-copilot/what-is-github-copilot`.

[5] A. Potdar and E. Shihab, “An Exploratory Study on Self-Admitted Technical Debt,” in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sep. 2014, pp. 91–100. DOI: `10.1109/ICSME.2014.31`. Accessed: Mar. 6, 2025. [Online]. Available: `https://ieeexplore.ieee.org/document/6976075`.

[6] D. OBrien, S. Biswas, S. M. Imtiaz, R. Abdalkareem, E. Shihab, and H. Rajan, “Are Prompt Engineering and TODO Comments Friends or Foes? An Evaluation on GitHub Copilot,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, Lisbon Portugal: ACM, Apr. 12, 2024, pp. 1–13, ISBN: 979-8-4007-0217-4. DOI: `10.1145/3597503.3639176`. Accessed: Dec. 10, 2024. [Online]. Available: `https://dl.acm.org/doi/10.1145/3597503.3639176`.

[7] GitHub, Inc. “GitHub · Build and ship software on a single, collaborative platform · GitHub,” Accessed: Feb. 18, 2025. [Online]. Available: `https://github.com/`.

[8] A. Sergeyuk, Y. Golubev, T. Bryksin, and I. Ahmed, “Using AI-based coding assistants in practice: State of affairs, perceptions, and ways forward,” *Information and Software Technology*, vol. 178, p. 107 610, Feb. 1, 2025, ISSN: 0950-5849. DOI: `10.1016/j.infsof.2024.107610`. Accessed: Dec. 10, 2024. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0950584924002155`.

[9] H. Hofmann and F. Lehner, “Requirements engineering as a success factor in software projects,” *IEEE Software*, vol. 18, no. 4, pp. 58–66, Jul. 2001, ISSN: 1937-4194. DOI: `10.1109/MS.2001.936219`. Accessed: Dec. 10, 2024. [Online]. Available: `https://ieeexplore.ieee.org/document/936219/?arnumber=936219`.

[10] D. OBrien, S. Biswas, S. Imtiaz, R. Abdalkareem, E. Shihab, and H. Rajan, *Replication package for “Are Prompt Engineering and TODO Comments Friends or Foes? An Evaluation on GitHub Copilot”*, Zenodo, Jan. 4, 2024. DOI: `10.5281/zenodo.10460738`. Accessed: Feb. 3, 2025. [Online]. Available: `https://zenodo.org/records/10460738`.

[11] S. Lauesen, *Software Requirements: Styles and Techniques*, Repr. London Munich: Addison-Wesley, 2008, 591 pp., ISBN: 978-0-201-74570-2.

[12] B. Davey and K. R. Parker, “Requirements Elicitation Problems: A Literature Analysis,” *Issues in Informing Science and Information Technology*, vol. 12, pp. 071–082, Jun. 3, 2015. Accessed: Feb. 20, 2025. [Online]. Available: `https://www.informingscience.org/Publications/2211`.

[13] C. Ebert and R. J. Wieringa, “Requirements Engineering: Solutions and Trends,” in *Engineering and Managing Software Requirements*, A. Aurum and C. Wohlin, Eds., Berlin, Heidelberg: Springer, 2005, pp. 453–476, ISBN: 978-3-540-28244-0. DOI: `10.1007/3-540-28244-0_20`. Accessed: Feb. 20, 2025. [Online]. Available: `https://doi.org/10.1007/3-540-28244-0_20`.

[14] D. Gobov and I. Huchenko, “Requirement Elicitation Techniques for Software Projects in Ukrainian IT: An Exploratory Study,” in *2020 15th Conference on Computer Science and Information Systems (FedCSIS)*, Sep. 2020, pp. 673–681. DOI: `10.15439/2020F16`. Accessed: Feb. 20, 2025. [Online]. Available: `https://ieeexplore.ieee.org/document/9222893`.

[15] S. Lim, A. Henriksson, and J. Zdravkovic, “Data-Driven Requirements Elicitation: A Systematic Literature Review,” *SN Computer Science*, vol. 2, no. 1, p. 16, Jan. 4, 2021, ISSN: 2661-8907. DOI: `10.1007/s42979-020-00416-4`. Accessed: Feb. 20, 2025. [Online]. Available: `https://doi.org/10.1007/s42979-020-00416-4`.

[16] T. Johann, C. Stanik, A. M. Alizadeh B., and W. Maalej, “SAFE: A Simple Approach for Feature Extraction from App Descriptions and App Reviews,” in *2017 IEEE 25th International Requirements Engineering Conference (RE)*, Sep. 2017, pp. 21–30. DOI: `10.1109/RE.2017.71`. Accessed: Mar. 3, 2025. [Online]. Available: `https://ieeexplore.ieee.org/document/8048887`.

[17] C. Pacheco and E. Tovar, “Stakeholder Identification as an Issue in the Improvement of Software Requirements Quality,” in *Advanced Information Systems Engineering*, J. Krogstie, A. Opdahl, and G. Sindre, Eds., Berlin, Heidelberg: Springer, 2007, pp. 370–380, ISBN: 978-3-540-72988-4. DOI: `10.1007/978-3-540-72988-4_26`.

[18] P. Kruchten, R. L. Nord, and I. Ozkaya, “Technical Debt: From Metaphor to Theory and Practice,” *IEEE Software*, vol. 29, no. 6, pp. 18–21, Nov. 2012, ISSN: 1937-4194. DOI: `10.1109/MS.2012.167`. Accessed: Feb. 26, 2025. [Online]. Available: `https://ieeexplore.ieee.org/document/6336722`.

[19] R. R. De Almeida, C. Treude, and U. Kulesza, “What’s behind tight deadlines? Business causes of technical debt,” in *2023 IEEE/ACM 16th International Conference on Cooperative and Human Aspects of Software Engineering (CHASE)*, May 2023, pp. 25–30. DOI: `10.1109/CHASE58964.2023.00011`. Accessed: Feb. 26, 2025. [Online]. Available: `https://ieeexplore.ieee.org/document/10164707`.

[20] N. Rios, R. Oliveira Spinola, M. G. de Mendonça Neto, and C. Seaman, “A Study of Factors that Lead Development Teams to Incur Technical Debt in Software Projects,” in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug. 2018, pp. 429–436. DOI: `10.1109/SEAA.2018.00076`. Accessed: Mar. 4, 2025. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/8498243`.

[21] N. S. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola, “Towards an Ontology of Terms on Technical Debt,” in *2014 Sixth International Workshop on Managing Technical Debt*, Sep. 2014, pp. 1–7. DOI: `10.1109/MTD.2014.9`. Accessed: Mar. 3, 2025. [Online]. Available: `https://ieeexplore.ieee.org/document/6974882`.

[22] S. Vaucher, F. Khomh, N. Moha, and Y.-G. Gueheneuc, “Tracking Design Smells: Lessons from a Study of God Classes,” in *2009 16th Working Conference on Reverse Engineering*, Oct. 2009, pp. 145–154. DOI: `10.1109/WCRE.2009.23`. Accessed: Mar. 4, 2025. [Online]. Available: `https://ieeexplore.ieee.org/document/5328806`.

[23] R. Ramač *et al.*, “Prevalence, common causes and effects of technical debt: Results from a family of surveys with the IT industry,” *Journal of Systems and Software*, vol. 184, p. 111 114, Feb. 1, 2022, ISSN: 0164-1212. DOI: `10.1016/j.jss.2021.111114`. Accessed: Feb. 26, 2025. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0164121221002119`.

[24] J. Tan, D. Feitosa, and P. Avgeriou, “The lifecycle of Technical Debt that manifests in both source code and issue trackers,” *Information and Software Technology*, vol. 159, p. 107 216, Jul. 1, 2023, ISSN: 0950-5849. DOI: `10.1016/j.infsof.2023.107216`. Accessed: Dec. 10, 2024. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0950584923000708`.

[25] J. P. Biazotto, D. Feitosa, P. Avgeriou, and E. Y. Nakagawa, “Technical debt management automation: State of the art and future perspectives,” *Information and Software Technology*, vol. 167, p. 107 375, Mar. 1, 2024, ISSN: 0950-5849. DOI: `10.1016/j.infsof.2023.107375`. Accessed: Mar. 3, 2025. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0950584923002306`.

[26] N. S. R. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, “Identification and management of technical debt: A systematic mapping study,” *Information and Software Technology*, vol. 70, pp. 100–121, Feb. 1, 2016, ISSN: 0950-5849. DOI: `10.1016/j.infsof.2015.10.008`. Accessed: Mar. 5, 2025. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0950584915001743`.

[27] S. Freire *et al.*, "Surveying Software Practitioners on Technical Debt Payment Practices and Reasons for not Paying off Debt Items," in *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '20, New York, NY, USA: Association for Computing Machinery, Apr. 17, 2020, pp. 210–219, ISBN: 978-1-4503-7731-7. DOI: `10.1145/3383219.3383241`. Accessed: Mar. 4, 2025. [Online]. Available: `https://dl.acm.org/doi/10.1145/3383219.3383241`.

[28] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" In *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 672–681. DOI: `10.1109/ICSE.2013.6606613`. Accessed: Mar. 6, 2025. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/6606613`.

[29] G. Szőke, C. Nagy, L. J. Fülöp, R. Ferenc, and T. Gyimóthy, "FaultBuster: An automatic code smell refactoring toolset," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sep. 2015, pp. 253–258. DOI: `10.1109/SCAM.2015.7335422`. Accessed: Mar. 6, 2025. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/7335422`.

[30] E. d. S. Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical Debt," in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, Oct. 2015, pp. 9–15. DOI: `10.1109/MTD.2015.7332619`. Accessed: Mar. 6, 2025. [Online]. Available: `https://ieeexplore.ieee.org/document/7332619`.

[31] E. Sutoyo and A. Capiluppi. "Self-Admitted Technical Debt Detection Approaches: A Decade Systematic Review." version 3. arXiv: `2312.15020 [cs]`, Accessed: Mar. 12, 2025. [Online]. Available: `http://arxiv.org/abs/2312.15020`, pre-published.

[32] Y. Li, M. Soliman, and P. Avgeriou, "Identification and Remediation of Self-Admitted Technical Debt in Issue Trackers," in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug. 2020, pp. 495–503. DOI: `10.1109/SEAA51224.2020.00083`. Accessed: Mar. 6, 2025. [Online]. Available: `https://ieeexplore.ieee.org/document/9226330`.

[33] Hewlett Packard Enterprise. "What is Open Source? | Glossary," Accessed: Mar. 12, 2025. [Online]. Available: `https://www.hpe.com/se/en/what-is/open-source.html`.

[34] GitHub, Inc. "About pull requests," GitHub Docs, Accessed: Mar. 12, 2025. [Online]. Available: `https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests`.

[35] A. Hars and S. Ou, "Working for Free? Motivations for Participating in Open-Source Projects," *International Journal of Electronic Commerce*, vol. 6, no. 3, pp. 25–39, 2002, ISSN: 1086-4415. JSTOR: `27751021`. Accessed: Mar. 13, 2025. [Online]. Available: `https://www.jstor.org/stable/27751021`.

[36] J. Coelho and M. T. Valente, "Why modern open source projects fail," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, New York, NY, USA: Association for Com-

puting Machinery, Aug. 21, 2017, pp. 186–196, ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3106246. Accessed: Mar. 12, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3106237.3106246.

[37] M. Woodward. "Octoverse 2022: 10 years of tracking open source," The GitHub Blog, Accessed: Mar. 14, 2025. [Online]. Available: https://github.blog/news-insights/research/octoverse-2022-10-years-of-tracking-open-source/.

[38] GitHub, Inc. "About Git," GitHub Docs, Accessed: Mar. 14, 2025. [Online]. Available: https://docs.github.com/en/get-started/using-git/about-git.

[39] Git. "Git-commit Documentation," Accessed: Feb. 14, 2025. [Online]. Available: https://git-scm.com/docs/git-commit.

[40] Git. "Git-show-ref Documentation," Accessed: Feb. 14, 2025. [Online]. Available: https://git-scm.com/docs/git-show-ref.

[41] GitHub, Inc. "About issues," GitHub Docs, Accessed: Mar. 14, 2025. [Online]. Available: https://docs.github.com/en/issues/tracking-your-work-with-issues/about-issues.

[42] GitHub, Inc. "REST API endpoints for issues," GitHub Docs, Accessed: Apr. 11, 2025. [Online]. Available: https://docs.github.com/en/rest/issues/issues?apiVersion=2022-11-28.

[43] C. Wanstrath. "Commit Comments," The GitHub Blog, Accessed: Apr. 11, 2025. [Online]. Available: https://github.blog/news-insights/commit-comments/.

[44] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "An in-depth study of the promises and perils of mining GitHub," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2035–2071, Oct. 1, 2016, ISSN: 1573-7616. DOI: 10.1007/s10664-015-9393-5. Accessed: Mar. 14, 2025. [Online]. Available: https://doi.org/10.1007/s10664-015-9393-5.

[45] M. Wessel *et al.*, "The Power of Bots: Characterizing and Understanding Bots in OSS Projects," *Proc. ACM Hum.-Comput. Interact.*, vol. 2, 182:1–182:19, CSCW Nov. 1, 2018. DOI: 10.1145/3274451. Accessed: Apr. 5, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3274451.

[46] GitHub, Inc. "Dependabot quickstart guide," GitHub Docs, Accessed: Apr. 2, 2025. [Online]. Available: https://docs.github.com/en/code-security/getting-started/dependabot-quickstart-guide.

[47] E. Tooley and E. Havens. "A smarter, quieter Dependabot," The GitHub Blog, Accessed: Apr. 2, 2025. [Online]. Available: https://github.blog/security/supply-chain-security/a-smarter-quieter-dependabot/.

[48] M. Golzadeh, T. Mens, A. Decan, E. Constantinou, and N. Chidambaram, "Recognizing Bot Activity in Collaborative Software Development," *IEEE Software*, vol. 39, no. 5, pp. 56–61, Sep. 2022, ISSN: 1937-4194. DOI: 10.1109/MS.2022.3178601. Accessed: Apr. 5, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9782582.

[49] GitHub, Inc. "Types of GitHub accounts," GitHub Docs, Accessed: Apr. 5, 2025. [Online]. Available: https://docs.github.com/en/get-started/learning-about-github/types-of-github-accounts.

[50] N. Chidambaram, A. Decan, and T. Mens, “A bot identification model and tool based on GitHub activity sequences,” *Journal of Systems and Software*, vol. 221, p. 112 287, Mar. 1, 2025, ISSN: 0164-1212. DOI: `10.1016/j.jss.2024.112287`. Accessed: Apr. 5, 2025. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0164121224003315`.

[51] GitHub, Inc. “Rate limits for the REST API,” GitHub Docs, Accessed: Apr. 5, 2025. [Online]. Available: `https://docs.github.com/en/rest/using-the-rest-api/rate-limits-for-the-rest-api?apiVersion=2022-11-28`.

[52] GitHub, Inc. “REST API endpoints for events,” GitHub Docs, Accessed: Apr. 5, 2025. [Online]. Available: `https://docs.github.com/en/rest/activity/events?apiVersion=2022-11-28`.

[53] GitHub, Inc. “GitHub Terms of Service,” GitHub Docs, Accessed: Apr. 10, 2025. [Online]. Available: `https://docs.github.com/en/site-policy/github-terms/github-terms-of-service`.

[54] GitHub, Inc. “No License,” Choose a License, Accessed: Apr. 22, 2025. [Online]. Available: `https://choosealicense.com/no-permission/`.

[55] XuehaiPan. “[Question & Bug] How to get license names for a dual-licensed repository with REST API · community · Discussion #40525,” GitHub, Accessed: Apr. 10, 2025. [Online]. Available: `https://github.com/orgs/community/discussions/40525`.

[56] GitHub, Inc. “REST API endpoints for licenses,” GitHub Docs, Accessed: Apr. 10, 2025. [Online]. Available: `https://docs.github.com/en/rest/licenses/licenses?apiVersion=2022-11-28`.

[57] Licensee. “Licensee,” Licensee, Accessed: Apr. 10, 2025. [Online]. Available: `https://licensee.github.io/licensee/`.

[58] Licensee. “Licensee/docs/what-we-look-at.md at main · licensee/licensee,” Accessed: Apr. 10, 2025. [Online]. Available: `https://github.com/licensee/licensee/blob/main/docs/what-we-look-at.md`.

[59] J. Kuriakose and J. Parsons, “How do open source software (OSS) developers practice and perceive requirements engineering? An empirical study,” in *2015 IEEE Fifth International Workshop on Empirical Requirements Engineering (EmpiRE)*, Aug. 2015, pp. 49–56. DOI: `10.1109/EmpiRE.2015.7431307`. Accessed: Dec. 10, 2024. [Online]. Available: `https://ieeexplore.ieee.org/document/7431307/?arnumber=7431307`.

[60] W. Scacchi, “Understanding Requirements for Open Source Software,” in *Design Requirements Engineering: A Ten-Year Perspective*, K. Lyytinen, P. Loucopoulos, J. Mylopoulos, and B. Robinson, Eds., Berlin, Heidelberg: Springer, 2009, pp. 467–494, ISBN: 978-3-540-92966-6. DOI: `10.1007/978-3-540-92966-6_27`.

[61] R. Vlas and W. N. Robinson, “A Rule-Based Natural Language Technique for Requirements Discovery and Classification in Open-Source Software Development Projects,” in *2011 44th Hawaii International Conference on System Sciences*, Jan. 2011, pp. 1–10. DOI: `10.1109/HICSS.2011.28`. Accessed: Mar. 13, 2025. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/5719011`.

[62] A. Potdar and E. Shihab. "Emad Shihab - SATD," Accessed: Mar. 15, 2025. [Online]. Available: `https://users.encs.concordia.ca/~eshihab/data/ICSME2014/satd.html`.

[63] T. Amanatidis, N. Mittas, A. Chatzigeorgiou, A. Ampatzoglou, and L. Angelis, "The Developer's Dilemma: Factors Affecting the Decision to Repay Code Debt," in *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)*, May 2018, pp. 62–66. Accessed: Mar. 3, 2025. [Online]. Available: `https://ieeexplore.ieee.org/document/8595105`.

[64] J. Tan, D. Feitosa, and P. Avgeriou, "Do practitioners intentionally self-fix Technical Debt and why?" In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2021, pp. 251–262. DOI: `10.1109/ICSME52107.2021.00029`. Accessed: Mar. 3, 2025. [Online]. Available: `https://ieeexplore.ieee.org/document/9609113`.

[65] Amazon Web Services, Inc. "What is LLM? - Large Language Models Explained," Accessed: Jan. 28, 2025. [Online]. Available: `https://aws.amazon.com/what-is/large-language-model/`.

[66] GitHub, Inc. "Quickstart for GitHub Copilot," GitHub Docs, Accessed: Mar. 11, 2025. [Online]. Available: `https://docs.github.com/en/copilot/quickstart`.

[67] Microsoft. "Visual Studio Code - Code Editing. Redefined," Accessed: Mar. 11, 2025. [Online]. Available: `https://code.visualstudio.com/`.

[68] G. Marvin, N. Hellen, D. Jjingo, and J. Nakatumba-Nabende, "Prompt Engineering in Large Language Models," in *Data Intelligence and Cognitive Informatics*, I. J. Jacob, S. Piramuthu, and P. Falkowski-Gilski, Eds., Singapore: Springer Nature, 2024, pp. 387–402, ISBN: 978-981-99-7962-2. DOI: `10.1007/978-981-99-7962-2_30`.

[69] GitHub, Inc. "GPT-4o Copilot: Your new code completion model is now generally available · GitHub Changelog," The GitHub Blog, Accessed: Mar. 29, 2025. [Online]. Available: `https://github.blog/changelog/2025-03-27-gpt-4o-copilot-your-new-code-completion-model-is-now-generally-available/`.

[70] M. S. Sheikhaei, Y. Tian, S. Wang, and B. Xu. "Understanding the Effectiveness of LLMs in Automated Self-Admitted Technical Debt Repayment." arXiv: `2501.09888 [cs]`, Accessed: Mar. 16, 2025. [Online]. Available: `http://arxiv.org/abs/2501.09888`, pre-published.

[71] H. Nunes, E. Figueiredo, L. Rocha, S. Nadi, F. Ferreira, and G. Esteves. "Evaluating the Effectiveness of LLMs in Fixing Maintainability Issues in Real-World Projects." arXiv: `2502.02368 [cs]`, Accessed: Mar. 16, 2025. [Online]. Available: `http://arxiv.org/abs/2502.02368`, pre-published.

[72] SonarSource SA. "Code Quality Tool & Secure Analysis with SonarQube," Accessed: Mar. 17, 2025. [Online]. Available: `https://www.sonarsource.com/products/sonarqube/`.

[73] X. Xiao, A. Lindberg, S. Hansen, and K. Lyytinen, ""Computing" Requirements for Open Source Software: A Distributed Cognitive Approach," *Journal of the Association for Information Systems*, vol. 19, no. 12, Dec. 28, 2018,

ISSN: 1536-9323. DOI: `10.17705/1jais.00525`. [Online]. Available: `https://aisel.aisnet.org/jais/vol19/iss12/2`.

[74] R. E. Vlas and W. N. Robinson, “Two Rule-Based Natural Language Strategies for Requirements Discovery and Classification in Open Source Software Development Projects,” *Journal of Management Information Systems*, vol. 28, no. 4, pp. 11–38, Apr. 1, 2012, ISSN: 0742-1222. DOI: `10.2753/MIS0742-1222280402`. Accessed: Mar. 17, 2025. [Online]. Available: `https://doi.org/10.2753/MIS0742-1222280402`.

[75] Slashdot Media. “Compare B2B Software, Download, & Develop Open Source & Business Software - SourceForge,” Accessed: Mar. 17, 2025. [Online]. Available: `https://sourceforge.net/`.

[76] GitHub, Inc. “About READMEs,” GitHub Docs, Accessed: Mar. 20, 2025. [Online]. Available: `https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-readmes`.

[77] T. Wang, S. Wang, and T.-H. (Chen, “Study the correlation between the readme file of GitHub projects and their popularity,” *Journal of Systems and Software*, vol. 205, p. 111 806, Nov. 1, 2023, ISSN: 0164-1212. DOI: `10.1016/j.jss.2023.111806`. Accessed: Mar. 20, 2025. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0164121223002017`.

[78] *PaddlePaddle/Paddle*, PaddlePaddle, May 5, 2025. Accessed: May 5, 2025. [Online]. Available: `https://github.com/PaddlePaddle/Paddle`.

[79] Creative Commons. “Deed - Attribution 4.0 International - Creative Commons,” Accessed: Apr. 10, 2025. [Online]. Available: `https://creativecommons.org/licenses/by/4.0/deed.en`.

[80] GitHub, Inc. “Using global node IDs,” GitHub Docs, Accessed: Apr. 22, 2025. [Online]. Available: `https://docs-internal.github.com/en/graphql/guides/using-global-node-ids`.

[81] GitHub, Inc. “GitHub Flavored Markdown Spec,” Accessed: Apr. 2, 2025. [Online]. Available: `https://github.github.com/gfm/`.

[82] avirshup. “Interactive simulation - Reply,” GitHub, Accessed: May 18, 2025. [Online]. Available: `https://github.com/Autodesk/molecular-design-toolkit/pull/129#discussion_r98984803`.

[83] P. Qi, Y. Zhang, Y. Zhang, J. Bolton, and C. D. Manning. “Stanza: A Python Natural Language Processing Toolkit for Many Human Languages.” arXiv: `2003.07082 [cs]`, Accessed: Apr. 15, 2025. [Online]. Available: `http://arxiv.org/abs/2003.07082`, pre-published.

[84] G. Grefenstette, “Tokenization,” in *Syntactic Wordclass Tagging*, H. van Halteren, Ed., Dordrecht: Springer Netherlands, 1999, pp. 117–133, ISBN: 978-94-015-9273-4. DOI: `10.1007/978-94-015-9273-4_9`. Accessed: Apr. 18, 2025. [Online]. Available: `https://doi.org/10.1007/978-94-015-9273-4_9`.

[85] M. P. Marcus, B. Santorini, and M. A. Marcinkiewicz, “Building a Large Annotated Corpus of English: The Penn Treebank,” *Computational Linguistics*, vol. 19, no. 2, J. Hirschberg, Ed., pp. 313–330, 1993. Accessed: Mar. 18, 2025. [Online]. Available: `https://aclanthology.org/J93-2004/`.

[86] Cambridge University Press & Assessment. “Modal verbs and modality,” Accessed: Apr. 19, 2025. [Online]. Available: `https://dictionary.cambridge.org/grammar/british-grammar/modal-verbs-and-modality`.

[87] shirady. “Git rename but GitHub shows it as added file and deleted file · community · Discussion #43177,” GitHub, Accessed: Apr. 20, 2025. [Online]. Available: `https://github.com/orgs/community/discussions/43177`.

[88] GitHub, Inc. “Enums,” GitHub Docs, Accessed: Apr. 19, 2025. [Online]. Available: `https://docs-internal.github.com/en/graphql/reference/enums`.

[89] GitHub, Inc. “GitHub Copilot - Visual Studio Marketplace,” Accessed: Apr. 22, 2025. [Online]. Available: `https://marketplace.visualstudio.com/items?itemName=GitHub.copilot`.

[90] Python Software Foundation. “PEP 8 – Style Guide for Python Code | peps.python.org,” Python Enhancement Proposals (PEPs), Accessed: Apr. 30, 2025. [Online]. Available: `https://peps.python.org/pep-0008/`.

[91] GitHub, Inc. “Configuring issue templates for your repository,” GitHub Docs, Accessed: May 5, 2025. [Online]. Available: `https://docs-internal.github.com/en/communities/using-templates-to-encourage-useful-issues-and-pull-requests/configuring-issue-templates-for-your-repository`.

[92] Z. Li, Y. Yu, T. Wang, Y. Lei, Y. Wang, and H. Wang, “To Follow or Not to Follow: Understanding Issue/Pull-Request Templates on GitHub,” *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2530–2544, Apr. 2023, ISSN: 1939-3520. DOI: `10.1109/TSE.2022.3224053`. Accessed: May 5, 2025. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/9961906`.

[93] OpenAI. “OpenAI Codex,” Accessed: May 5, 2025. [Online]. Available: `https://openai.com/index/openai-codex/`.

[94] GitHub, Inc. “GPT-4o Copilot (Codex V2) code completion model [GA] · Issue #1114 · github/roadmap,” GitHub, Accessed: May 5, 2025. [Online]. Available: `https://github.com/github/roadmap/issues/1114`.

[95] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, “The promises and perils of mining GitHub,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, Hyderabad India: ACM, May 31, 2014, pp. 92–101, ISBN: 978-1-4503-2863-0. DOI: `10.1145/2597073.2597074`. Accessed: Jan. 27, 2025. [Online]. Available: `https://dl.acm.org/doi/10.1145/2597073.2597074`.

[96] V. Käfer, D. Graziotin, I. Bogicevic, S. Wagner, and J. Ramadani, “Poster: Communication in Open-Source Projects–End of the E-mail Era?” In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, May 2018, pp. 242–243. Accessed: Jan. 30, 2025. [Online]. Available: `https://ieeexplore.ieee.org/document/8449520`.

[97] Mozilla Corporation. “401 Unauthorized - HTTP | MDN,” Accessed: May 5, 2025. [Online]. Available: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status/401`.

[98] Mozilla Corporation. “404 Not Found - HTTP | MDN,” Accessed: May 5, 2025. [Online]. Available: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status/404`.

[99] Internet Archive. “Wayback Machine,” Accessed: May 5, 2025. [Online]. Available: `https://web.archive.org/`.

[100] GitHub, Inc. “Disabling issues,” GitHub Docs, Accessed: Feb. 5, 2025. [Online]. Available: `https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/enabling-features-for-your-repository/disabling-issues`.

[101] Stanford NLP Group. “Available Models & Languages,” Stanza, Accessed: May 5, 2025. [Online]. Available: `https://stanfordnlp.github.io/stanza/available_models.html`.

[102] Stanford NLP Group. “Language Identification,” Stanza, Accessed: May 5, 2025. [Online]. Available: `https://stanfordnlp.github.io/stanza/langid.html`.

[103] Python Software Foundation. “Ast — Abstract Syntax Trees,” Python documentation, Accessed: Mar. 16, 2025. [Online]. Available: `https://docs.python.org/3/library/ast.html`.

[104] A. Shiervani, “Dataset for "prompt engineering with requirements in GitHub copilot",” May 2025. DOI: `10.6084/m9.figshare.29172113.v1`. [Online]. Available: `https://figshare.com/articles/dataset/Dataset_for_Prompt_Engineering_with_Requirements_in_GitHub_Copilot_/29172113`.

A

Appendix

A.1 Source Code

requirements.txt

```
# Torch with CUDA 12.6
--extra-index-url https://download.pytorch.org/whl/cu126

requests==2.32.3
stanza==1.10.1
torch==2.6.0
transformers==4.49.0
pandas==2.2.3
PyGithub==2.6.1
python-dotenv==1.0.1
ordered-set==4.1.0
humanize==4.12.2
ruff==0.11.2
requests-cache==1.2.1
zstandard==0.23.0
pymongo==4.11.3
diskcache==5.6.3
numpy==2.2.4
matplotlib==3.10.1
```

filter.py

```python
import os
import logging
import json
from io import BytesIO
from zipfile import ZipFile
from ordered_set import OrderedSet
import requests
import pandas as pd
from github import Github
```

```python
from github import Auth
from github.Repository import Repository
from github.ContentFile import ContentFile
from github.GithubException import UnknownObjectException
from dotenv import load_dotenv
from stanza.pipeline.core import Pipeline
from stanza.pipeline.core import DownloadMethod

logger = logging.getLogger(__name__)


def main():
    zip_name = "./CopilotTODO.zip"

    logging.basicConfig(
        filename="filter.log",
        level=logging.INFO,
        filemode="w",
        format="%(asctime)s [%(levelname)s] %(name)s"
        + " %(filename)s:%(funcName)s %(message)s",
        force=True,
    )

    original_no_fixes = read_ids_from_csv(zip_name, "no_fixes.csv")
    original_yes_fixes = read_ids_from_csv(zip_name, "yes_fixes.csv")
    original = original_no_fixes.union(original_yes_fixes)
    logger.info("Original IDs: %i rows", len(original))

    english_only, repo_links = filter_english_repos(original, zip_name)
    logger.info("Only English IDs: %i rows", len(english_only))

    no_expired = filter_expired(english_only, zip_name, repo_links)
    logger.info("Not expired TODO comments:\n%s", no_expired)

    process_licenses(no_expired)


def process_licenses(df: pd.DataFrame):
    csv_name = "repo_licenses.csv"

    if os.path.exists(csv_name):
        licenses_df = pd.read_csv(csv_name)

        only_permissive = licenses_df[licenses_df["Permissive"].eq("Yes")][
            "Repository"
        ].to_list()
        to_keep = df[df["Repository"].isin(only_permissive)]
        to_keep.to_csv("filtered.csv", index=False)

        original_count = len(df.index)
        kept_count = len(to_keep.index)

        logger.info(
```

```
            "Total %i of %i TODO comments have been removed "
            + "due to being in a repository without a permissive license. "
            + "Remaining %i comments were stored in filtered.csv.",
            original_count - kept_count,
            original_count,
            kept_count,
        )
    else:
        unique_repos = OrderedSet(df["Repository"])
        with open(csv_name, "w") as f:
            f.write("Repository,Permissive,Explanation\n")
            for unique_repo in unique_repos:
                f.write(unique_repo + ",?,?\n")

        logger.info(
            "Created repo_licenses.csv. "
            + "This file should be filled in to create filtered.csv."
        )


def read_ids_from_csv(archive_name: str, csv_name: str):
    csv_name = f"CopilotTODO/data_scripts/{csv_name}"

    if not os.path.isfile(archive_name):
        raise FileNotFoundError(archive_name)

    df = None

    with ZipFile(archive_name) as z:
        if csv_name not in z.NameToInfo:
            raise LookupError(archive_name, csv_name)

        with BytesIO(z.read(csv_name)) as content:
            df = pd.read_csv(content)

    return OrderedSet(df["Unnamed: 0"].array.tolist())


def filter_english_repos(cids: OrderedSet, archive_name: str):
    csv_name = "CopilotTODO/data_scripts/2-sample.txt"

    repo_cids = {}
    with ZipFile(archive_name) as z:
        if csv_name not in z.NameToInfo:
            raise LookupError(archive_name, csv_name)

        content = z.read(csv_name).decode("utf-8").splitlines()

        counter = 0
        for row in content:
            counter += 1
            if counter not in cids:
                continue
```

```

            row_split = row.split(",")
            repo = row_split[0].removeprefix("https://github.com/")

            repos = repo_cids.keys()
            if repo not in repos:
                repo_cids[repo] = set()

            repo_cids[repo].add(counter)

    to_drop = set()
    repo_links = {}

    load_dotenv()
    if "GITHUB_API_TOKEN" not in os.environ:
        raise ValueError(".env missing GITHUB_API_TOKEN")
    auth = Auth.Token(os.getenv("GITHUB_API_TOKEN"))
    with Github(auth=auth, per_page=100) as g:
        nlp = Pipeline(
            lang="multilingual",
            processors="langid",
            download_method=DownloadMethod.REUSE_RESOURCES,
        )

        repos = repo_cids.keys()
        repo: str
        for repo in repos:
            gh_repo = None
            readme = None
            try:
                gh_repo = g.get_repo(repo)
                readme = gh_repo.get_readme()

                if isinstance(readme, ContentFile):
                    readme = readme.decoded_content.decode("utf-8")
            except UnknownObjectException:
                pass

            processed = None
            if isinstance(readme, str):
                processed = nlp(readme)

                if processed.lang != "en":
                    logger.warning(
                        "The language of the README in repository `%s` "
                        + "was `%s` instead of `en`",
                        repo,
                        processed.lang,
                    )
            else:
                logger.warning("Could not find README for repository `%s`", repo)

            # Most of the NLDs in this repository is in Chinese.
```

```
            if repo == "PaddlePaddle/Paddle":
                processed = None

            if processed is None or processed.lang != "en":
                for cid in repo_cids[repo]:
                    to_drop.add(cid)
            elif isinstance(gh_repo, Repository):
                current_url = gh_repo.html_url.removeprefix("https://github.com/")
                if repo != current_url:
                    logger.warning(
                        "Detected repository move from path `%s` to `%s`",
                        repo,
                        current_url,
                    )

                parent_repo = ""
                if gh_repo.fork:
                    parent_repo = gh_repo.parent.html_url.removeprefix(
                        "https://github.com/"
                    )
                    logger.warning(
                        "Repository `%s` was detected as a fork of repository `%s`",
                        current_url,
                        parent_repo,
                    )

                repo_links[repo] = [current_url, parent_repo]
            else:
                raise AssertionError(repo)

    cids = cids.difference(to_drop)

    return cids, repo_links


def filter_expired(cids: OrderedSet, archive_name: str, repo_links: map):
    file_name = "CopilotTODO/data_scripts/2-sample.txt"

    cids_repofile = {}
    original_todos = {}
    with ZipFile(archive_name) as z:
        if file_name not in z.NameToInfo:
            raise LookupError(archive_name, file_name)

        content = z.read(file_name).decode("utf-8").splitlines()

        counter = 0
        for row in content:
            counter += 1
            if counter not in cids:
                continue

```

```
            row_split = row.split(",", maxsplit=2)
            repo = row_split[0].removeprefix("https://github.com/")
            file = row_split[1]
            todo = str(json.loads(f'"{row_split[2]}"'))

            cids_repofile[counter] = [repo, file]
            original_todos[counter] = todo

    to_drop = set()

    load_dotenv()
    if "GITHUB_API_TOKEN" not in os.environ:
        raise ValueError(".env missing GITHUB_API_TOKEN")
    auth = Auth.Token(os.getenv("GITHUB_API_TOKEN"))
    with Github(auth=auth, per_page=100) as g:
        for cid, repofile in cids_repofile.items():
            repo = repo_links[repofile[0]][0]
            file = repofile[1]
            original_todo = original_todos[cid]

            contents = None
            try:
                gh_repo = g.get_repo(repo)
                contents = gh_repo.get_contents(path=file)

                if isinstance(contents, ContentFile):
                    link = contents.html_url
                    if contents.encoding == "none":
                        # weird edge case, gotta download directly.
                        logger.warning(
                            "Edge case encountered when downloading `%s`
    through API",
                            link,
                        )
                        r = requests.get(link, timeout=30)
                        if r.ok:
                            contents = r.text
                        else:
                            raise AssertionError(repo, file, r)
                    else:
                        contents = contents.decoded_content.decode("utf-8"
    )
            except UnknownObjectException:
                pass

            contains_todo = False
            if isinstance(contents, str):
                contains_todo = original_todo.lower() in contents.lower()

            if not contains_todo:
                if isinstance(link, str):
                    logger.warning(
                        "File `%s` no longer contains: `%s`", link,
    original_todo
```

```python
                        )
                    to_drop.add(cid)

    cids = cids.difference(to_drop)

    df = pd.DataFrame(columns=("ID", "Repository", "Filename", "Content"))
    for i, cid in enumerate(cids):
        cid_repofile = cids_repofile[cid]
        repo_link = repo_links[cid_repofile[0]]
        repository = repo_link[0]

        # Should be empty for all repos
        parent = repo_link[1]
        if parent:
            raise AssertionError(repository, parent)

        filename = cid_repofile[1]
        content = original_todos[cid]
        df.loc[i] = [cid, repository, filename, content]

    return df


if __name__ == "__main__":
    main()
```

download_repos.py

```
File has been truncated. Click on the title box to access the attachment.
```

finalize.py

```python
from collections import OrderedDict
from json import loads
import logging
import os
import shutil
from ordered_set import OrderedSet
import zstandard
import pandas as pd

logger = logging.getLogger(__name__)


def main():
    csv_name = "./filtered.csv"
    output_dir = os.path.abspath("./repos/")

    logging.basicConfig(
```

```
        filename="finalize.log",
        level=logging.INFO,
        filemode="w",
        format="%(asctime)s [%(levelname)s] %(name)s"
        + " %(filename)s:%(funcName)s %(message)s",
        force=True,
    )

    filtered = pd.read_csv(csv_name)
    repo_strs = OrderedSet(sorted(filtered["Repository"].to_list()))

    repo_stats: OrderedDict = {
        repo: {
            "preprocessed": False,
            "issues": 0,
            "prs": 0,
            "commits": 0,
            "commits_pr": 0,
            "documents": 0,
            "documents_issues": 0,
            "documents_prs": 0,
            "documents_commits": 0,
            "documents_parent": 0,
            "documents_fork": 0,
        }
        for repo in repo_strs
    }

    for parent_repo in repo_stats.keys():
        value = repo_stats[parent_repo]
        repo_path = os.path.join(output_dir, parent_repo)

        preprocessed_path = os.path.join(repo_path, "PREPROCESSED")
        excluded_path = os.path.join(repo_path, "EXCLUDED")

        if not os.path.exists(preprocessed_path):
            continue
        elif os.path.exists(excluded_path):
            continue
        else:
            value["preprocessed"] = True

        repo_stats[parent_repo] = get_stats(parent_repo, output_dir, value
    )

    original_comment_sample, no_code_sample, sampled_repo =
    get_comment_samples(
        output_dir, repo_stats
    )

    to_save = [
        repo
        for repo, value in repo_stats.items()
        if (value["preprocessed"] and value["documents"] > 0)
```

```
    ]
    finalized = filtered[filtered["Repository"].isin(to_save)].sort_values("ID")
    finalized.to_csv("finalized.csv", index=False)
    logger.info(
        "Resulting dataset of %i TODO documents, found in %i repositories, "
        + "was stored in finalized.csv.",
        len(finalized.index),
        len(to_save),
    )

    no_documents = [
        repo_str
        for repo_str in repo_strs
        if (repo_str not in to_save and repo_stats[repo_str]["preprocessed"])
    ]
    for repo_str in no_documents:
        value = repo_stats[repo_str]
        repo_path = os.path.join(output_dir, repo_str)

        excluded_path = os.path.join(repo_path, "EXCLUDED")
        if not os.path.exists(excluded_path):
            repo_files = [
                os.path.join(repo_path, repo_file)
                for repo_file in [
                    "commits.json.zst",
                    "issues.json.zst",
                    "metadata.json.zst",
                    "code.zip",
                    "requirements.json.zip",
                ]
            ]

            for repo_file in repo_files:
                if os.path.isfile(repo_file):
                    os.remove(repo_file)

            todo_files_dir = os.path.join(repo_path, "todo_files")
            if os.path.isdir(todo_files_dir):
                shutil.rmtree(todo_files_dir)

            with open(excluded_path, "x", encoding="utf-8") as f:
                f.write(repo_str)

    logger.info(
        "Excluded %i preprocessed repositories in total because they had no documents. "
        + "Additionally, their downloaded files were removed to preserve disk space.",
        len(no_documents),
    )

```

```
not_preprocessed = [
    repo_str for repo_str in repo_strs if not repo_stats[repo_str]["preprocessed"]
]
logger.info(
    "%i repositories are still pending preprocessing.",
    len(not_preprocessed),
)

one, two, three, four, five = 0, 0, 0, 0, 0
for repo in to_save:
    comment_count = repo_stats[repo]["documents"]
    num = int(comment_count)
    if num > 0 and num <= 10:
        one += 1
    elif num > 10 and num <= 100:
        two += 1
    elif num > 100 and num <= 1000:
        three += 1
    elif num > 1000 and num <= 10000:
        four += 1
    elif num > 10000:
        five += 1
logger.info(
    "Counts for graph: "
    + "(1--10, %i) (10--100, %i) (100--1000, %i) (1000--10000, %i) (>10000, %i)",
    one,
    two,
    three,
    four,
    five,
)

assert original_comment_sample
with open("sample-original_comment.txt", "w", newline="") as f:
    f.write(original_comment_sample)

assert no_code_sample
with open("sample-no_code.txt", "w", newline="") as f:
    # lstinputlisting ignores last line if ends with DOS newline
    if no_code_sample.endswith("\r\n"):
        no_code_sample += "\r\n"
    f.write(no_code_sample)

logger.info("Sampled comment from repository %s.", sampled_repo)

stats_csv = "./finalized_stats.csv"
with open(stats_csv, "w") as f:
    f.write(
        "Repository,Is Preprocessed,Issues,PRs,Commits,PR Commits,All Documents,"
        + "Issue Documents,PR Documents,Commit Documents,Parent Documents,"
```

```
            + "Fork Documents\n"
        )
        for repo, value in repo_stats.items():
            str_values = [str(value_value) for value_value in value.values()]
            f.write(f"{repo},{','.join(str_values)}\n")


def get_stats(parent_repo: str, output_dir: str, value: dict[str,]):
    repo_path = os.path.join(output_dir, parent_repo)

    all_issues_documents: dict[str, list] = {}
    issues_path = os.path.join(repo_path, "issues.json.zst")
    with zstandard.open(issues_path, "rb", encoding="utf-8") as f:
        all_issues_documents = loads(f.readall())

    for repo_str, issues_documents in all_issues_documents.items():
        current_issue_documents, current_pr_documents = 0, 0
        for issue_documents in issues_documents:
            documents_count = len(issue_documents["comments"])

            # Count body as a comment (because it is also an NLD)
            if issue_documents["no_code"]:
                documents_count += 1

            if "pull_request" in issue_documents:
                current_pr_documents += documents_count
                value["prs"] += 1
            elif "issue" in issue_documents:
                current_issue_documents += documents_count
                value["issues"] += 1

        value["documents_issues"] += current_issue_documents
        value["documents_prs"] += current_pr_documents

        total_documents = current_issue_documents + current_pr_documents

        if parent_repo == repo_str:
            value["documents_parent"] += total_documents
        else:
            value["documents_fork"] += total_documents

    parent_commits = []
    commits_path = os.path.join(repo_path, "commits.json.zst")
    with zstandard.open(commits_path, "rb", encoding="utf-8") as f:
        parent_commits = loads(f.readall())

    for commit_documents in parent_commits:
        documents_len = len(commit_documents["comments"])

        value["documents_commits"] += documents_len
        value["documents_parent"] += documents_len
        value["commits"] += 1

```

```
    value["documents"] = (
        value["documents_issues"] + value["documents_prs"] + value["documents_commits"]
    )
    return value


def get_comment_samples(output_dir: str, repo_stats: dict[str, dict[str,]]):
    for parent_repo, values in repo_stats.items():
        if not values["preprocessed"] or values["documents"] == 0:
            continue

        repo_path = os.path.join(output_dir, parent_repo)
        all_issues_documents: dict[str, list] = {}
        issues_path = os.path.join(repo_path, "issues.json.zst")
        with zstandard.open(issues_path, "rb", encoding="utf-8") as f:
            all_issues_documents = loads(f.readall())

        for repo_str, issues_documents in all_issues_documents.items():
            for issue_documents in issues_documents:
                for comment in issue_documents["comments"]:
                    raw_body = str(comment["raw"]["body"])
                    lines = raw_body.count("\r\n") + raw_body.count("\n\n")
                    if lines < 5:
                        no_code = str(comment["no_code"])
                        if raw_body != no_code:
                            return raw_body, no_code, repo_str

    return "", "", ""


if __name__ == "__main__":
    main()
```

elicit_reqs.py

```
File has been truncated. Click on the title box to access the attachment.
```

select_reqs.py

```
File has been truncated. Click on the title box to access the attachment.
```

gen_prompts.py

```python
from json import loads
import logging
import os
from pathlib import Path
import pandas as pd


logger = logging.getLogger(__name__)


def main():
    csv_name = "./with_reqs.csv"
    output_dir = os.path.abspath("./repos/")

    logging.basicConfig(
        filename="gen_prompts.log",
        level=logging.INFO,
        filemode="w",
        format="%(asctime)s [%(levelname)s] %(name)s"
        + " %(filename)s:%(funcName)s %(message)s",
        force=True,
    )

    with_reqs = pd.read_csv(csv_name)
    gen_prompts(output_dir, with_reqs)
    logger.info("Done writing prompts for %i TODO comments", len(with_reqs.index))


def gen_prompts(output_dir: str, df: pd.DataFrame):
    no_req_grps = (
        "There is a TODO comment stating '{0}' in the current script '{1}'.\n"
        + "Provide code which solves the issue that is described by this TODO comment."
    )
    one_req_grp = (
        "There is a TODO comment stating '{0}' in the current script '{1}'.\n"
        + "\n"
        + "Below is one group of requirement documents related to this comment:\n"
        + "{2}\n"
        + "\n"
        + "Provide code which solves the issue that is described by this TODO comment. "
        + "Use the specified group of requirement documents as a source of context."
    )
    three_req_grps = (
        "There is a TODO comment stating '{0}' in the current script '{1}'.\n"
```

```
    + "\n"
    + "Below is three groups of requirement documents related to this
comment:\n"
    + "This is the first group:\n{2}\n"
    + "This is the second group:\n{3}\n"
    + "This is the third group:\n{4}\n"
    + "\n"
    + "Provide code which solves the issue that is described by this
TODO comment. "
    + "Use the specified groups of requirement documents as a source
of context."
)

for _, row in df.iterrows():
    todo_id = str(row["ID"])
    todo_content = row["Content"]
    todo_filename = row["Filename"]
    todo_basename = os.path.basename(todo_filename)
    todo_stemname = Path(todo_basename).stem
    repo_str = row["Repository"]

    req_grps = []

    repo_path = os.path.join(output_dir, repo_str)
    todo_files_dir = os.path.join(repo_path, "todo_files")
    todo_output_dir = os.path.join(todo_files_dir, todo_id)
    req_output_file = os.path.join(todo_output_dir, "requirements.json
")
    with open(req_output_file, "r", encoding="utf-8") as in_file:
        req_grps = loads(in_file.read())

    todo_prompts = [
        no_req_grps.format(todo_content, todo_basename),
        one_req_grp.format(
            todo_content,
            todo_basename,
            format_req_grp(req_grps[0]),
        ),
        three_req_grps.format(
            todo_content,
            todo_basename,
            format_req_grp(req_grps[0]),
            format_req_grp(req_grps[1]),
            format_req_grp(req_grps[2]),
        ),
    ]

    script_content = ""
    todo_input_file = os.path.join(todo_output_dir, todo_basename)
    with open(todo_input_file, "r", encoding="utf-8", newline="") as
in_file:
        script_content = in_file.read()

    for index, todo_prompt in enumerate(todo_prompts):
```

```python
            prompt_file = os.path.join(
                todo_output_dir, f"{todo_stemname}_rq{index + 2}.txt"
            )
            with open(prompt_file, "w", encoding="utf-8", newline="") as out_file:
                out_file.write(
                    f"-----Input-----\n\n{todo_prompt}"
                    + "\n\n-----Output-----\n\n\n\n"
                    + "-----Resolved (Yes/Partially/No)-----\n\n\n\n"
                    + "-----Comments-----\n\n"
                )

            todo_output_file = os.path.join(
                todo_output_dir, f"{todo_stemname}_rq{index + 2}.py"
            )
            with open(todo_output_file, "w", encoding="utf-8", newline="") as out_file:
                out_file.write(script_content)


def format_req_grp(grp):
    req_grp_inner = (
        "Title: '{0}'. Group type: '{1}'. Relevance: '{2}'.\nRequirements:\n{3}\n"
    )
    req_inner = "Type: '{0}'. Importance: '{1}'. Content:\n{2}\n"

    return req_grp_inner.format(
        grp["title"],
        grp["type"],
        human_relevance(grp["relevance"]["value"]),
        "\n".join(
            [
                req_inner.format(
                    text["type"],
                    human_importance(text["importance"]),
                    text["text"],
                )
                for text in grp["texts"]
            ]
        ),
    )


def human_importance(importance: float):
    if importance == 1:
        return "Most important"
    elif importance == 2 / 3:
        return "Important"
    elif importance == 1 / 3:
        return "Somewhat important"
    elif importance == 0:
        return "Least important"

```

```
    assert False


def human_relevance(relevance: float):
    if relevance == 1.0:
        return "Most relevant"
    elif relevance == 0.5:
        return "Somewhat relevant"

    assert False


if __name__ == "__main__":
    main()
```

prompt_stats.py

```
import logging
import os
from pathlib import Path
import numpy as np
import pandas as pd


logger = logging.getLogger(__name__)


def main():
    csv_name = "./with_reqs.csv"
    output_dir = os.path.abspath("./repos/")

    logging.basicConfig(
        filename="prompt_stats.log",
        level=logging.INFO,
        filemode="w",
        format="%(asctime)s [%(levelname)s] %(name)s"
        + " %(filename)s:%(funcName)s %(message)s",
        force=True,
    )

    with_reqs = pd.read_csv(csv_name)

    todos_prompted = get_prompted(output_dir, with_reqs)
    todos_prompted.to_csv("todos_prompted.csv", index=False)

    print_stats(todos_prompted)


def get_prompted(output_dir: str, df: pd.DataFrame):
    extra_info = pd.DataFrame(
        data="",
```

```
        index=np.arange(len(df.index)),
        columns=[
            "Has Exception",
            "Exception Comments",
            "RQ 2 Resolution",
            "RQ 2 Comments",
            "RQ 3 Resolution",
            "RQ 3 Comments",
            "RQ 4 Resolution",
            "RQ 4 Comments",
        ],
    )
    todos_prompted = df.join(extra_info)

    for index, row in df.iterrows():
        todo_id = str(row["ID"])
        todo_filename = row["Filename"]
        todo_basename = os.path.basename(todo_filename)
        todo_stemname = Path(todo_basename).stem
        repo_str = row["Repository"]

        repo_path = os.path.join(output_dir, repo_str)
        todo_files_dir = os.path.join(repo_path, "todo_files")
        todo_output_dir = os.path.join(todo_files_dir, todo_id)

        todo_index = todos_prompted.index[index]
        for rqi in range(2, 5):
            prompt_file = os.path.join(todo_output_dir, f"{todo_stemname}_rq{rqi}.txt")

            output, resolved, comments = "", "", ""

            with open(prompt_file, "r", encoding="utf-8") as out_file:
                next_output, next_resolved, next_comments = False, False, False
                for line in out_file:
                    line = line.replace("\n", "")
                    line = line.replace("\r", "")
                    if not line:
                        continue

                    if next_comments:
                        comments += line
                    elif line == "-----Resolved (Yes/Partially/No)-----":
                        next_resolved = True
                    elif line == "-----Comments-----":
                        next_comments = True
                    elif line == "-----Output-----":
                        next_output = True
                    elif next_resolved:
                        resolved += line
                    elif next_output:
                        output += line

```

```
                assert next_output and next_resolved and next_comments
                assert output
                assert resolved

                if output == "N/A":
                    assert comments
                    todos_prompted.loc[todo_index, "Has Exception"] = "Yes"
                    todos_prompted.loc[todo_index, "Exception Comments"] = comments
                    break

            if resolved not in ["Yes", "Partially", "No"]:
                raise ValueError(resolved)

            todos_prompted.loc[todo_index, f"RQ {rqi} Resolution"] = resolved
            todos_prompted.loc[todo_index, f"RQ {rqi} Comments"] = comments

    return todos_prompted


def print_stats(df: pd.DataFrame):
    rq2_success = len(df[df["RQ 2 Resolution"] == "Yes"].index)
    rq2_partial = len(df[df["RQ 2 Resolution"] == "Partially"].index)
    rq2_fail = len(df[df["RQ 2 Resolution"] == "No"].index)
    logger.info(
        "Counts for `rq2 resolution` graph: " + "(Yes, %i) (Partially, %i) (No, %i)",
        rq2_success,
        rq2_partial,
        rq2_fail,
    )

    rq3_success = len(df[df["RQ 3 Resolution"] == "Yes"].index)
    rq3_partial = len(df[df["RQ 3 Resolution"] == "Partially"].index)
    rq3_fail = len(df[df["RQ 3 Resolution"] == "No"].index)
    logger.info(
        "Counts for `rq3 resolution` graph: " + "(Yes, %i) (Partially, %i) (No, %i)",
        rq3_success,
        rq3_partial,
        rq3_fail,
    )

    rq4_success = len(df[df["RQ 4 Resolution"] == "Yes"].index)
    rq4_partial = len(df[df["RQ 4 Resolution"] == "Partially"].index)
    rq4_fail = len(df[df["RQ 4 Resolution"] == "No"].index)
    logger.info(
        "Counts for `rq4 resolution` graph: " + "(Yes, %i) (Partially, %i) (No, %i)",
        rq4_success,
        rq4_partial,
```

```
        rq4_fail,
    )

    rq2_with_comments = df[~df["RQ 2 Comments"].eq("")]
    rq2_combined: list[str] = (
        "- Type: RQ 2. Resolution: "
        + rq2_with_comments["RQ 2 Resolution"]
        + ". Comments:\n"
        + rq2_with_comments["RQ 2 Comments"]
        + "\n"
    ).to_list()

    rq3_with_comments = df[~df["RQ 3 Comments"].eq("")]
    rq3_combined: list[str] = (
        "- Type: RQ 3. Resolution: "
        + rq3_with_comments["RQ 3 Resolution"]
        + ". Comments:\n"
        + rq3_with_comments["RQ 3 Comments"]
        + "\n"
    ).to_list()

    rq4_with_comments = df[~df["RQ 4 Comments"].eq("")]
    rq4_combined: list[str] = (
        "- Type: RQ 4. Resolution: "
        + rq4_with_comments["RQ 4 Resolution"]
        + ". Comments:\n"
        + rq4_with_comments["RQ 4 Comments"]
        + "\n"
    ).to_list()

    rqs_combined = sorted(rq2_combined + rq3_combined + rq4_combined)
    logger.info(
        "All %i comments:\n------------\n\n%s\n------------",
        len(rqs_combined),
        "\n".join(rqs_combined),
    )

    exceptions = len(df[df["Has Exception"] == "Yes"].index)
    logger.info("Total of %i TODOs were not prompted due to exceptions.", exceptions)
    exception_comments = set(df["Exception Comments"].to_list())
    logger.info(
        "All exception comments:\n------------\n%s\n\n------------",
        "\n- ".join(exception_comments),
    )


if __name__ == "__main__":
    main()
```

A.2 Data

Dataset published on `figshare` [104].