Investigating Using LLM for Close-Coding Tasks on Software Engineering Domain

Robiul Islam
miislam@wm.edu
William and Mary
Williamsburg, Virginia, USA

Abstract

Artificial Intelligence (AI) rapidly transforms many areas, including education, healthcare, and daily life. A major driver of this change is Large Language Models(LLMs), powerful AI systems capable of understanding and generating human-like text. In this study, we explore how LLMs can support qualitative research in software engineering. Qualitative research involves analyzing non-numerical data such as interview transcripts or open-ended survey responses to identify patterns and themes. In this study, we focus on a method known as closed coding, where researchers apply predefined codes to the data to systematically identify key ideas. To help LLMs perform close coding, we use prompt engineering, the practice of carefully designing questions or instructions to get accurate and useful responses from the model. Our results show that LLMs can effectively assist with close coding, potentially making qualitative analysis in software engineering faster and more efficient.

ACM Reference Format:
Robiul Islam. 2025. Investigating Using LLM for Close-Coding Tasks on Software Engineering Domain. In . ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Large Language Models (LLMs) have been used for different sectors, from healthcare to education. Historically, software development was a manual process involving multiple stages such as requirements gathering, hand-coding, debugging, testing, and deployment all performed predominantly by humans without significant automation. However, new developments in powerful language models have introduced the possibility of automating numerous tasks across different areas of software development, such as bug fixing [36], code summarization [35], code completion [21], and assert statement generation [31]. Software engineering increasingly uses LLMs because they offer a new way to handle tasks, essentially reframing them as problems of examining information, programs, and written material. LLMs demonstrate significant promise in revolutionizing how numerous software engineering activities are carried out. Within the field of software engineering, when dealing with non-numerical information, language models are becoming crucial tools. These models can greatly speed up the process of finding and understanding key information from extensive amounts of written material, effectively turning a time-consuming task into an automated procedure. The conventional method for analyzing information relies heavily on skilled individuals, leading to prolonged timelines and substantial labor investments. LLMs are being applied to interpret non-numerical data across various research areas, including human-computer interaction [12]. Traditional methods of data analysis rely heavily on human judgment and expertise. However, the rise of LLMs marks a significant shift toward more automated and intelligent analysis systems. These models can process large volumes of qualitative data, such as user feedback, software documentation, and development logs, to uncover meaningful insights. As a result, LLMs not only make the analysis process faster and more efficient but also allow for more detailed and nuanced interpretations of the data, which were often difficult to achieve with manual methods. Numerous researchers have investigated how LLMs can be applied to the analysis of qualitative data. Xiao et al. examine the use of LLMs, such as GPT-3, to support deductive qualitative coding with pre-defined codebooks, without the need for model fine-tuning [33]. Their findings indicate that LLM-generated codes show fair to substantial agreement with expert annotations, demonstrating the potential of LLMs in qualitative analysis. However, the study does not explore why the model fails to generate certain codes, nor does it provide a comparative analysis of errors made by human annotators versus those made by the model. Similar to previous work, Rasheed et al. investigate the application of LLMs to automate and streamline qualitative data analysis in Software Engineering (SE), aiming to reduce the time and effort required for manual coding [25]. While their multi-agent LLM framework demonstrated improvements in processing efficiency and

Conference'17, Washington, DC, USA

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/10.1145/nnnnnnn.nnnnnnn

scalability, the study does not provide a detailed analysis of the model's limitations, such as the reasons behind coding errors or a comparison of failure cases between the model and human annotators. Thus, we identify a current research gap in understanding why LLMs generate incorrect codes or fail to assign any code to certain survey responses.

Our study enhances the understanding of the limitations and challenges in applying LLMs for qualitative coding, highlights key areas where future research and tool development should focus, and offers a detailed discussion of potential strategies to address these challenges.

To summarize, this paper makes the following key contributions:

- We present a multi-agent framework leveraging LLaMA-based language models to support qualitative close coding for open-ended survey questions, using pre-defined codebooks to evaluate survey responses in Software Engineering.
- Our study provides an analysis of coding failures by both human annotators and LLMs, identifying cases where the model generated incorrect codes or failed to assign codes.
- A replication package [23] containing the prompts, models, scripts, and documentation has been provided to support future research in this area.

2 BACKGROUND AND RELATED WORK

In this section, we briefly present the related work of the study with a focus on existing research.

2.1 Large Language Models in Software Engineering

In recent years, LLMs have shown significant application in various Software Engineering research [7]. LLMs, such as ChatGPT [22], Llama [19], Gemini [9], are designed to produce text that closely mimics natural human language [6]. GPT models stand out for their versatility in handling various language tasks, including translation, summarization, question-answering, and creative writing, all without requiring specialized training for each task [25]. GPT is a leading example of LLMs, demonstrating their potential in diverse applications that demand sophisticated language comprehension and generation [30]. LLMs show significant improvement over traditional program synthesis on code generation [16], code summarization [1], data analysis [24], text classification [4], software development [26], code search [11], unit test case generation [29], automated program repair [18], etc. The LLMs are trained on vast amounts of source code and natural language; these models can understand, generate, and manipulate code in various programming languages and human-written text. Unlike traditional program synthesis approaches, which depend on formal specifications or domain-specific logic, LLMs are data-driven and capable of generalizing from examples. Their ability to process both code and natural language makes them especially useful in real-world software development, where developer intent is often expressed informally. Despite their promise, LLMs also raise concerns related to code correctness, security vulnerabilities, explainability, and alignment with developer intent, which are active areas of research.

2.2 Large Language Models in Qualitative Research

Researchers across various disciplines rely on qualitative methods, particularly qualitative coding, which involves labeling and categorizing textual data, to achieve their research objectives. This form of coding is distinct from computer programming and refers instead to the analysis of non-numerical data, such as interview transcripts or closed-ended survey responses. Qualitative coding enables scholars to analyze and interpret complex phenomena within their respective fields systematically. Many such studies gather audio or video data, such as recordings of interviews, focus groups, or consultations, which are typically transcribed into text for detailed analysis [2]. Qualitative data, encompassing non-numeric information such as text, interviews, and observations, often requires extensive manual analysis. Processing large qualitative datasets in this manner is time-consuming and inefficient, highlighting the need for more automated and scalable approaches. In this era of technological advancements, there are plenty of software tools available, such as ATLAS.ti [13], MAXDQA [17], and NVivo [5] designed for qualitative data analysis. These tools, while effective for manual qualitative analysis, become inefficient and time-consuming when applied to large datasets. They lack modern, AI-driven automation that could enhance coding accuracy, speed, and scalability. As the LLMs draw attention to qualitative research, several researchers have used LLMs for qualitative research, such as Bernerd et al., in their exploration of using LLMs like ChatGPT for qualitative research. They discussed several ways in which ChatGPT can significantly expedite the process of analyzing qualitative data. Traditional qualitative research methods, like interviews and open-ended surveys, are often time-consuming, especially when it comes to tasks like transcription and coding, making them less efficient for large datasets. By leveraging the power of LLMs, the researchers demonstrated how tasks that typically take weeks can now be completed in a matter of hours [15]. Jie Gao et al. proposed CollabCoder. CollabCoder can provide code suggestions, also it can help to develop a codebook from scratch [8]. Ziang et al. found that instead of fine-tuning, it is possible to use prompt engineering for qualitative analysis. They found the performance shifted when they used one-shot learning instead of zero-shot learning [34]. Robert et al. conducted interviews and collected the data, then they created a codebook and input the sample text and codebook into an LLM. Furthermore, they asked the LLM to determine if the provided text is present in the sample text, and if that is present, they asked for the evidence

[28]. In the paper [3], Barany et al. compare four different approaches to codebook development: a fully manual method with no ChatGPT involvement, a fully automated method without human input, and two hybrid approaches where humans contribute either at the beginning or the end of the process. They found that whether GPT participates early or last produces more reliable code and is rated better than humans. They also found that there is little difference between the hybrid approach and the human approach. They found that the fully automated which is ChatGPT, can have a huge impact on improving the quality of the codebook, but still human involvement is important. In the paper [32], Alexander et al. conducted two types of interviews: one between humans and another between humans and AI. Their study, involving university students in politics, revealed that AI conversational interviews generated higher-quality data comparable to traditional approaches. Zeeshan et al. introduced an LLM-based multi-agent to automate various types of qualitative data analysis [25]. In this study, we aim to develop a system that can automate or partially automate open-coding tasks, specifically assigning codes to survey responses. To achieve this, we will use various models (such as Llama3-8B, Llama3-70B, and Llama3-70B-Instruct) and technique prompt engineering. Unlike some prior research, which primarily focuses on generating key theme from the survey response, our study centers on the annotation of survey responses. We utilize advanced models and techniques that are more likely to yield accurate results. While earlier studies have applied LLMs for coding based on predefined codebooks, our approach introduces more flexibility and provides a detailed analysis of why the model might fail to assign the correct code.

3 STUDY METHODOLOGY

The primary goal of this study is to explore how LLMs can support qualitative research methods—specifically, close coding in the context of software engineering. Traditionally, assigning codes to open-ended survey responses is a time-consuming task that requires careful human judgment. We aim to investigate whether LLMs can help automate this process while maintaining consistency and relevance. To conduct this study, we use a diverse set of survey responses, where participants answered domain specific questions related to software engineering. Trevor et al. [27], conducted on a survey among 138 practitioners across five distinct stakeholder groups: those experienced with SBOMs, members of key open-source projects, professionals in AI/ML, experts in cyber-physical systems, and legal practitioners. Differentiated questionnaires were used for each group to gather tailored insights. We selected 8 survey questions, each receiving 20 responses, resulting in a total of 160 responses for analysis. The details of the dataset are provided in subsection 3.1.

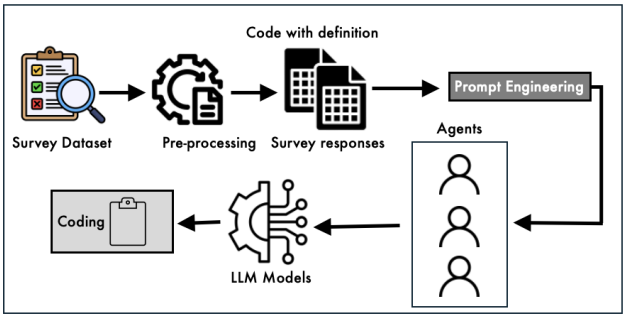

Figure 1. Proposed system's workflow for qualitative data analysis

Each response was then evaluated to determine whether it matched a predefined code based on its corresponding definition. We employed a multi-agent prompting approach, where three separate instances of an LLM were used, each configured with a distinct prompt. Each model (agent) was asked to assess whether a given survey response clearly and directly aligned with a specific code definition. If out of nine combinations, six were yes, then we assigned that code to the response. We use Ollama [20] to send both the prompt and the selected model at the same time, allowing it to return the generated response. This method allowed us to compare the behavior of differently prompted models, reduce individual model bias, and explore the reliability of LLMs in supporting close coding tasks.

This study aims to address the following research questions (RQs):

RQ_1: How many times do the models assign the same code as the human annotator?

RQ_2: In how many cases do the models either assign extra codes or fail to assign the same code as the human annotator?

RQ_3: In how many cases do the models generate at least one code that is similar to the code written by the human annotator?

RQ_4: Is it always the case that the human annotator assigns the most appropriate code, or can the LLaMA model sometimes assign a better-fitting code based on the code definition?

RQ_5: In a setup using 3 agents and 3 prompts (for a total of 9 decisions per response), how does a higher number of 'yes' responses affect the chance of an exact match with human-coded results?

In Figure 1, we present a detailed overview of our strategy, highlighting the main steps and their interactions.

3.1 Dataset

The dataset used in this study consists of real-world survey responses related to the topic of Software Bill of Materials

Table 1. Number of codes for each survey question

No.	Survey Question	Number of code
1	Q1	23
2	Q2	20
3	Q3	8
4	Q4	19
5	Q5	15
6	Q6	20
7	Q7	19
8	Q8	22

(SBOMs) [27]. It aggregates the results of five distinct surveys conducted across various domains, including machine learning and cyber-physical systems. These surveys were designed to capture diverse perspectives on SBOM practices, challenges, and implementation strategies. The dataset includes a variety of response types, such as multiple-choice questions (MCQs) and text-only responses, enabling both quantitative and qualitative analysis. We specifically focused on text-based responses, where users shared their thoughts regarding the questions. Our analysis concentrated on the machine learning and cyber-physical system questions, as each of these had 20 responses. We excluded other questions with fewer responses, such as those with only 6. Table 1 shows the number of codes assigned to each question. Each question received the same number of responses, which is 20.

Before analysis, the responses were anonymized to preserve participant confidentiality. Minimal preprocessing, including correcting typographical errors and normalizing formatting in free-text answers, is performed to improve the consistency and accuracy of the data without altering its meaning. This ensures that the responses are clean and uniform, making them easier to analyze while preserving the semantic content. Without this step, inconsistencies like spelling mistakes or formatting variations could introduce noise, potentially affecting the reliability and validity of the analysis. We consider the data that includes responses from all participants. We collected the responses to seven questions from Machine Learning, and one from both Critical Projects. For both categories, there are 20 responses for each question. Some survey responses with the human annotator with the llama-generated are available in Table 2. Out of 20 responses, 18 responses are shown there from the subset of the dataset machine learning, and the question asked the responders is: What are some of the benefits you expect to see from using AIBOMs, if any?. The survey response and the human codes present in the Table 2.

3.2 Pre-processing

Before conducting the analysis, the survey data underwent several preprocessing steps to ensure consistency, clarity, and usability. The preprocessing process was designed to preserve the original meaning of the responses while minimizing noise and irrelevant variation. This study focuses only on open-ended (free-text) responses; closed-ended questions such as multiple-choice (MCQs) were not considered. First, any empty responses were removed from the dataset. Then, we cleaned the text by removing unnecessary characters such as double quotation marks (e.g., converting "clear" to clear), newline characters (\n), and extra spaces. This preprocessing step improved tokenization, prevented parsing errors, and enhanced the efficiency of the analysis, ensuring more accurate and reliable results. All preprocessing steps were carefully chosen to preserve the original meaning of the responses while ensuring the dataset was clean, consistent, and ready for reliable analysis.

3.3 Prompt Engineering

Prompt engineering is the process of designing and refining input queries to guide the behaviour of LLMs such as Llama3 [10]. Since LLMs are sensitive to input phrasing and structure, the way a prompt is crafted can significantly influence the quality, relevance, and reliability of the model's output. Effective prompt engineering involves clarifying task instructions, providing examples (few-shot prompting) [12], or encouraging step-by-step reasoning (e.g., chain-of-thought prompting) [14]. We initially started with a single prompt where we provided the survey response, the assigned code, and the definition of the code. The model was asked to extract key themes from the survey response and compare them with the code. If any of the identified themes matched the code, the model would return 'yes,' and the code would be considered relevant for that specific response. However, the results from this approach were not satisfactory, leading us to explore alternative techniques. Figure 3 presents the initial prompt; however, we later refined this method for improved accuracy and performance.

3.4 Agents

Previously, we used a single prompt to compare each survey response with a code based on its definition. However, the results were not consistent. To improve this, we adopted a multi-agent approach, as shown in Figure 4. In this setup, we use three different LLaMA models: LLaMA3-70B, LLaMA3-70B-Instruct, and LLaMA3-8B.

Each model receives three different prompt versions, making a total of nine combinations, as illustrated in Figure 2. The three different prompt versions for each model were derived through an iterative process aimed at improving the accuracy and relevance of the model's responses. Initially, a basic prompt was created to assess the model's ability to extract key themes and match them with the predefined codes. Based on the performance of this initial prompt, adjustments were made to refine the structure, clarity, and specificity of the prompts. These changes aimed to better guide the model

Table 2. Survey responses with human codes

Survey response	Human codes
Increased reproducibility, easier iteration on previous works, more accessible resources, earlier detection of weaknesses/vulnerabilities.	['Security', 'OpenAccess', 'Reproducability', 'ImprovedIteration']
It will provide structure to help people document their work.	['BetterDocumentation']
Similar answer to DataBOMs.	['Reproducibility', 'IncreasedTransparency']
This would help developers not involved in the project have a better understanding of the software they are working with.	['Teaching']
Unsure, not experienced enough with them.	['Unsure']
Idealy I don't have to implement features like versioing manually myself, and even get inspired to use techniques that I usually don't think of myself (e.g. I would have never written code to monitor the GPU memory usage or GPU compute utility while training, even if I didn't have a tool that would monitor it for me.) and therefore ensure that I am using best practices or to be more concrete, my colleagues with lesser experience use best practices without me having to tell them everything.	['BestPractices', 'Automation']
An example of usage for the model.	['ExampleModelUsage']
Reproducibility.	['Reproducability']
Reducing security risks. Making deployment/updates faster.	['Security', 'FasterDeployment']
As with DataBOMs, one big benefit is the ability to better detect biases and reproduce results.	['Reproducability', 'DetectBiases']
The secure and safe usage in critical domains.	['Security']
Trustworthy – against supply chain type of attacks.	['Security', 'Trust']
As for the DataBOM, it would help increase the trust in the model.	['Trust']
No comment	['BadAnswer']
Might help with the so-called "reproducibility crisis" in AI, though frankly I'm skeptical. I think the bigger benefit would be a CVE-like reporting of bias in datasets that let us see which models should be updated.	['Reproducability', 'DetectBiases']
Continous Integration of AI models.	['ContinuousIntegration']
Similarly to dataBOMs, it could help with reproducing studies, better collaboration, faster iteration and improvements of model. Bugs, biases, and fairness issues could be discovered quickly.	['ProblemLocation', 'ImprovedIteration', 'DetectBiases', 'Reproducability', 'BetterCollaboration']
First and foremost, help manage compliance risk.	['Compliance']

Table 3. Codes and Its definitions

Code	Definition
Reproducability	AIBOMs make it easier to reproduce work.
Security	AIBOMs make it easier to detect/locate vulnerabilities or mitigate other security risks.
OpenAccess	AIBOMs make it faster/easier to access resources associated with a system.
BetterDocumentation	AIBOMs assist with documenting work.
IncreasedTransparency	Increased transparency into how the data was acquired, processed, etc.
Teaching	AIBOMs can help people unfamiliar with the project learn more about it.
Unsure	Answer equivalent to I don't know
Automation	AIBOMs help to automate development tasks.
BestPractices	The use of AIBOMs helps developers to follow best practices.
ExampleModelUsage	The AIBOM provides an example of usage for the model.
FasterDeployment	AIBOMs help developers make updates and deployments faster.
DetectBiases	AIBOMs make it easier to identify potential biases in the model.
ProblemLocation	AIBOMs highlight issues with models and make them easier to find.
VerifyUsability	Help to verify usability of the system. (Vague answer)
Trust	AIBOMs increase trust in the model.
ImprovedIteration	AIBOMs will make the iterative process of improving on previous works much easier
ContinuousIntegration	AIBOMs will facilitate the continuous integration of ai models
BadAnswer	Response is not applicable or nonsensical.
BetterCollaboration	AIBOM could make collaboration easier
Compliance	AIBOMs will assist in managing compliance

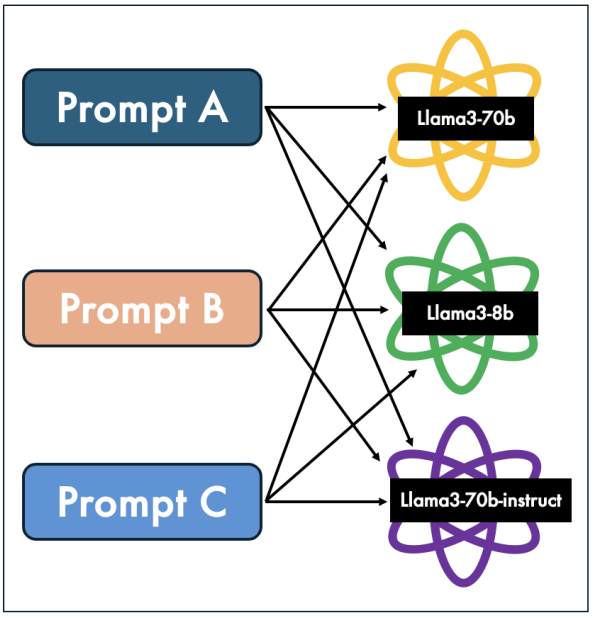

Figure 2. Multi-agent system for verifying survey response

in understanding the context of the survey responses and the definitions of the codes.

For each model, the prompts were carefully tailored to ensure that the language used was precise and aligned with the model's strengths. Different versions were tested, and the most effective prompts were selected based on the models' performance in matching survey responses to the correct codes.

Every prompt includes four elements: the survey question, the survey response, the code, and the code definition. The model is then asked to decide if the survey response matches the code based on its definition.

The threshold of "six" out of nine model-prompt combinations returning a "yes" response was chosen to ensure a robust consensus among the models. This approach minimizes the likelihood of errors or biases from any single model or prompt combination and ensures that the assigned code has strong support across multiple perspectives. By setting the threshold at six, we allow for some flexibility in case of minor inconsistencies or ambiguities in individual model responses while still ensuring that the majority of models agree on the appropriateness of the code for the survey response. This helps balance accuracy and reliability in the final coding decision.

4 RESULTS

Our study aims to accurately assign the most appropriate code to each survey response. Assigning one or more codes to a response helps summarize its content and contributes to a broader understanding of the overall survey findings. In our approach, each prompt given to the language model includes the survey response, a candidate code, the corresponding code definition, and the question that was asked to the responder. We use the different LLaMA models to determine

```
prompt = f"""
    [INST] <<SYS>>
      Given this code: {code}, the definition of the code: {definition},
      and the survey response: {survey_text}

      - Extract the key themes from the survey response.
      - Compare the key themes with the given code.

      Rules for answering:
      - Answer "yes" only if any key theme strongly match or 100% with the given code.
      - Answer "no" if code does not strongly match with any of the key theme.
      - Answer "no" if the key theme partially match with the code.
      - Do not assume missing themes.
      - Return only "yes" or "no"
      - Avoid any explanation

    <</SYS>> [/INST]
    """
```

Figure 3. Prompt Engineering Technique-1

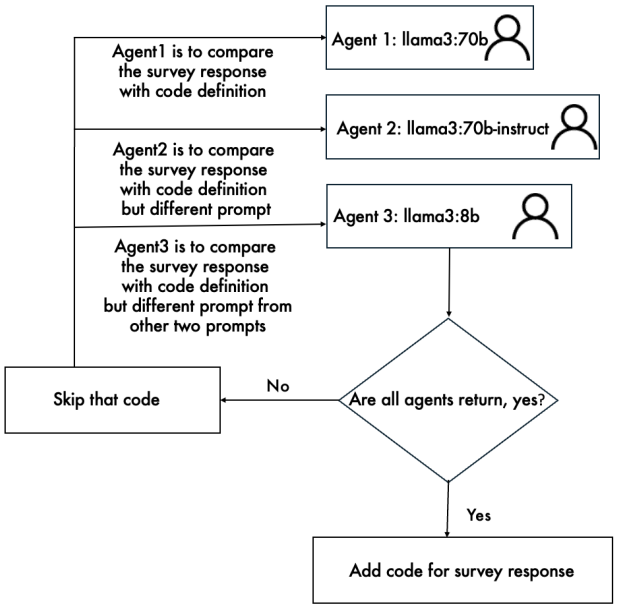

Figure 4. Decision making criteria for a survey response

whether the given code accurately represents the survey response based on the provided definition. The model selects from a predefined codebook. For each survey response, the model evaluates all codes in the codebook individually. If the model responds "yes" to a specific code, that code is assigned to the response. However, since the model can occasionally assign incorrect codes, it is important to assess how well it performs this task. To evaluate the model's accuracy in code assignment, we apply one performance metric. In particular, we focus on the Exact Match metric, which measures whether the set of codes assigned by the model exactly matches the set of codes assigned by human annotators.

Exact Match: Exact Match (EM) is a straightforward and effective metric for evaluating the performance of LLMs in classification tasks like code assignment. In our study, EM is particularly suitable because the model does not generate new codes it simply checks whether a given code from the codebook matches a survey response based on the code's definition. This eliminates common issues such as case sensitivity or partial overlaps. For instance, if the correct code is "WM," then "wm" would not exist in the codebook, avoiding mismatches due to capitalization. Each survey response is evaluated against all codes in the codebook. If the model identifies a match (i.e., responds "yes"), that code is assigned to the response. After processing, we compare the model-assigned codes with the ground truth labels provided by human annotators. EM measures how often the model's set of assigned codes exactly matches the human-assigned set. Given the structured nature of our task selecting codes from a fixed codebook EM serves as a precise and reliable metric to assess the model's effectiveness in replicating human coding behavior.

4.1 RQ_1: How model generate the same code as a human annotator?

To evaluate the consistency of code assignments using LLMs, we measured the frequency of exact matches between survey responses and predefined code definitions. Figure 5 visualizes the exact match results. The highest number of exact matches occurred in Q7, with 11 matches, or 55%, indicating strong alignment between model responses and the code definitions in this area. However, Q5 showed 0 exact matches.

This variation suggests that the performance of the models may be influenced by factors such as the clarity and specificity of the code definitions, as well as the complexity of the topics discussed in the survey responses. In the case of Q7, the model was likely able to identify clear, well-defined themes that matched the predefined codes, resulting in higher accuracy. On the other hand, Q5 may have contained more ambiguous or less clearly defined concepts, making it harder for the model to make precise matches. Additionally, the domain of the question, such as whether it relates to well-established fields like machine learning, could also play a role in the consistency of model alignment. Topics with more structured and standardized language may lead to more accurate and consistent code assignments, while less clearly defined or more complex topics may introduce variability in model performance. This variation underscores the importance of having well-defined codebook categories and clear, unambiguous survey questions to achieve more reliable model outcomes.

In Table 4, we present the codes generated by the model for each survey response. The order of the codes is not considered when comparing with human-coded responses, as it typically depends on the predefined codebook structure. However, when only one code is assigned, the model's output must match the human code exactly, such as a response labeled "Trust" needing to be identified as "Trust" by the model. Additionally, the table shows the verdict of the Llama-generated code: the blue color (●) indicates that the model

Table 4. Survey information with LLaMA generated code

Survey response	Human codes	Llama generated	Verdict
Increased reproducibility, easier iteration on previous works, more accessible resources, earlier detection of weaknesses/vulnerabilities.	['Security', 'OpenAccess', 'Reproducability', 'ImprovedIteration']	['Reproducability', 'Security', 'ImprovedIteration']	Model failed to assign code
It will provide structure to help people document their work.	['BetterDocumentation']	['BetterDocumentation']	Exact Match
Similar answer to DataBOMs.	['Reproducibility', 'IncreasedTransparency']	['BadAnswer']	Human annotator failed
This would help developers not involved in the project have a better understanding of the software they are working with.	['Teaching']	['BetterDocumentation', 'Teaching', 'BetterCollaboration']	Model assign wrong code
Unsure, not experienced enough with them.	['Unsure']	['Unsure']	Exact Match
An example of usage for the model.	['ExampleModelUsage']	['ExampleModelUsage', 'BadAnswer']	Model assign wrong code
Reproducibility.	['Reproducability']	['Reproducability']	Exact Match
Reducing security risks. Making deployment/updates faster.	['Security', 'FasterDeployment']	['Security', 'FasterDeployment']	Exact Match
As with DataBOMs, one big benefit is the ability to better detect biases and reproduce results.	['Reproducability', 'DetectBiases']	['DetectBiases']	Model failed to assign code
The secure and safe usage in critical domains.	['Security']	['Trust']	Model assign wrong code
Trustworthy – against supply chain type of attacks.	['Security', 'Trust']	['Security', 'Trust']	Exact Match
As for the DataBOM, it would help increase the trust in the model.	['Trust']	['Trust']	Exact Match
No comment	['BadAnswer']	['BadAnswer']	Exact Match
Might help with the so-called "reproducibility crisis" in AI, though frankly I'm skeptical. I think the bigger benefit would be a CVE-like reporting of bias in datasets that let us see which models should be updated.	['Reproducability', 'DetectBiases']	[]	Model failed to assign code
Continous Integration of AI models.	['ContinuousIntegration']	['Automation', 'ContinuousIntegration']	Model failed
Similarly to dataBOMs, it could help with reproducing studies, better collaboration, faster iteration and improvements of model. Bugs, biases, and fairness issues could be discovered quickly.	['ProblemLocation', 'ImprovedIteration', 'DetectBiases', 'Reproducability', 'BetterCollaboration']	['Reproducability', 'ImprovedIteration', 'BetterCollaboration']	Model failed to assign code
First and foremost, help manage compliance risk.	['Compliance']	['Compliance']	Exact Match

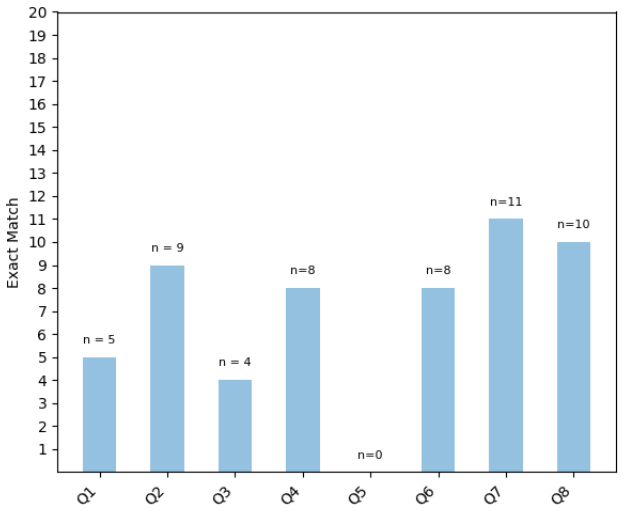

Figure 5. Exact matches for different questions

generated a code similar to the one assigned by the human annotator; the orange color (🟠) represents cases where the human annotator failed to assign the correct code; and the red color (🔴) indicates that the model either assigned the wrong code or failed to generate code.

> **Summary of RQ$_1$:** Our study shows that out of 160 survey responses, the LLaMA models produced exact matches for 55 responses, resulting in an exact match rate of approximately 34%.

4.2 RQ$_2$: How many cases model added new code or fail to assign the code?

The results indicate that the models frequently failed to assign the correct codes to survey responses. As shown in Figure 6, the model demonstrated limited accuracy in several instances. For the machine learning question: "`How can we ensure that AIBOMs completely and correctly report all the dependencies of ML/DL systems?`, the model correctly assigned the expected code in only 5 out of 20 responses. In the remaining 15 cases, it either introduced new codes not present in the human-coded ground truth or omitted relevant codes. Notably, the model failed to assign 11 expected codes across these responses. In another question: `What are some of the benefits you expect to see from using AIBOMs, if any?`, the model achieved 9 correct code assignments out of 20. However, in 7 of the remaining 11 responses, it generated codes that were not included in the ground truth. For critical project question: `How do your projects obtain the list of dependencies? Are there any specific tools or techniques used to that end?`, The model correctly assigned codes in 4 out of 20 responses. It introduced new codes in 14 cases and failed to generate any code in 6 responses. We examined the most frequently added and omitted codes to further analyze these discrepancies. The model incorrectly added the code BadAnswer for the machine learning-related questions in 6 out of 20 responses. In the critical project responses, the model added the code ToolAssisted in 4 instances. Additionally, the model consistently failed to assign the code ReproduceModel in the machine learning context, and PackageFile in the critical project context, despite their presence in the human-coded annotations. These findings highlight significant gaps between model-generated codes and human annotations, particularly in terms of both false positives (added codes) and false negatives (missed codes).

> **Summary of RQ$_2$:** The results show that the models often failed to assign the correct codes to survey responses. The model often added or removed code for survey responses.

4.3 RQ$_3$: How many cases model added at least one code which is present in the human-written code?

As illustrated in Figure 6, the model frequently generated codes that were not present in the ground-truth, or it frequently failed to generate code. In this context, a "removed" code refers to a situation where a code present in the ground truth, such as BadAnswer, is not produced by the model, and instead a different, unrelated code, such as ReproduceModel, is generated. In some cases, the model removed the correct code without generating any new code in its place. Across the 160 survey responses analyzed, each response was annotated with a minimum of one and a maximum of six codes in the ground truth. Out of the 160 responses, the model achieved exact matches in 55 cases. Among the remaining 105 cases, we further examined partial overlaps and found that in 47 of these, the model generated at least one code that matched a code in the ground truth. In Figure 7, shown that for the survey question Q_3, the model generates 4 exact matches. However, the model generates at least one code 14 times, which are present in the ground truth. This suggests that while exact agreement was limited, the model was still partially aligned with human annotations in most cases. The model failed to generate an exact match due to the code definitions and the way the human annotator assigned the codes.

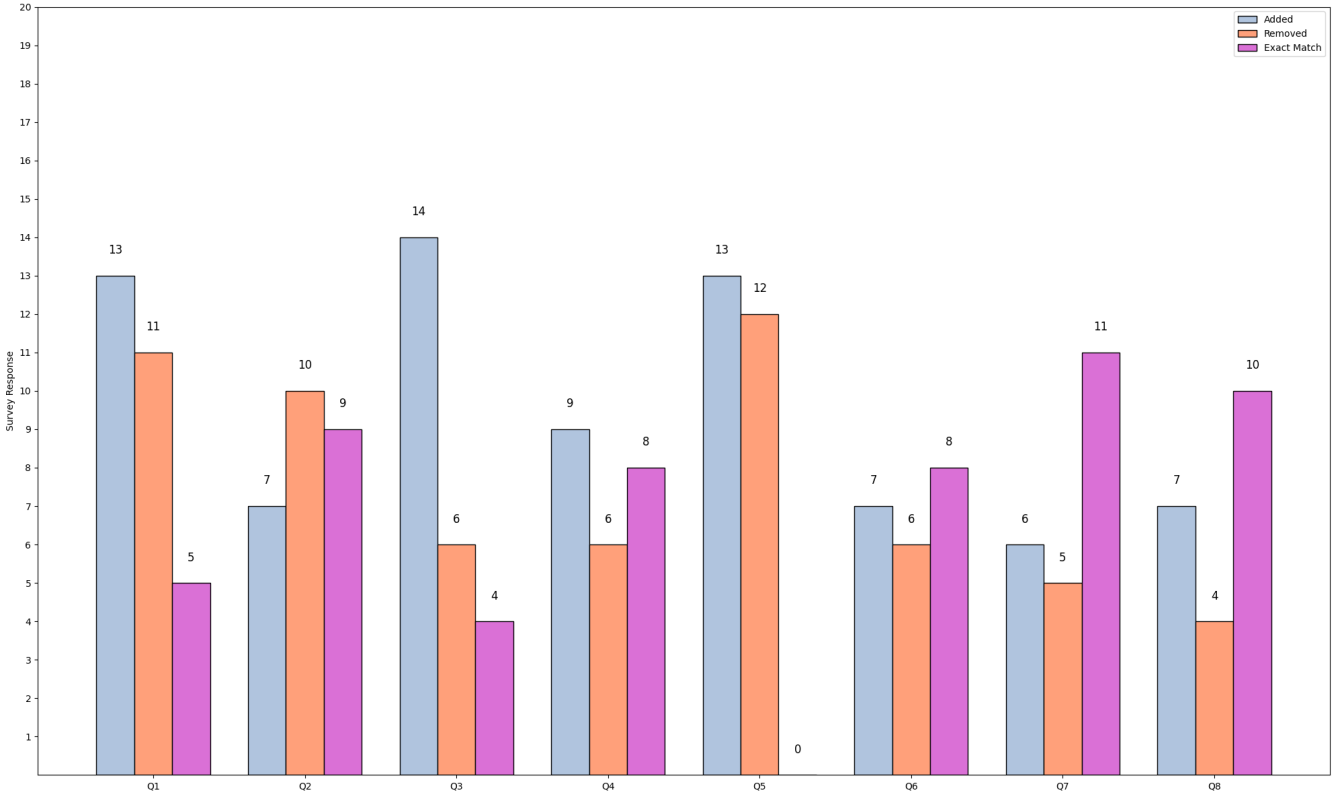

Figure 6. Analysis of coding errors: additions, omissions, and matches

> **Summary of RQ$_3$:** The analysis reveals that the model frequently failed to generate the correct set of codes, either omitting expected ones or introducing unrelated codes. Out of 160 survey responses, exact matches with human annotations were found in only 55 cases. However, in 47 of the remaining 105 responses, the model produced at least one correct code from the ground truth. This indicates that while full agreement was limited, the model showed partial alignment with human coding in many cases.

4.4 RQ$_4$: How many cases did the model generate the perfect code but the human annotator failed ?

In this section, we examine instances where the model assigned a code based on the provided definition, but the corresponding code was not included in the human-annotated ground truth. To better understand these discrepancies, we manually analyzed all survey responses where the model-generated code differed from the human annotations. As illustrated in Figure 6, the model more frequently added new codes rather than omitted expected ones. This raises the question of whether these additions indicate genuine errors by the model or oversights by the human annotators. Table 5 presents several examples of such cases, where the model

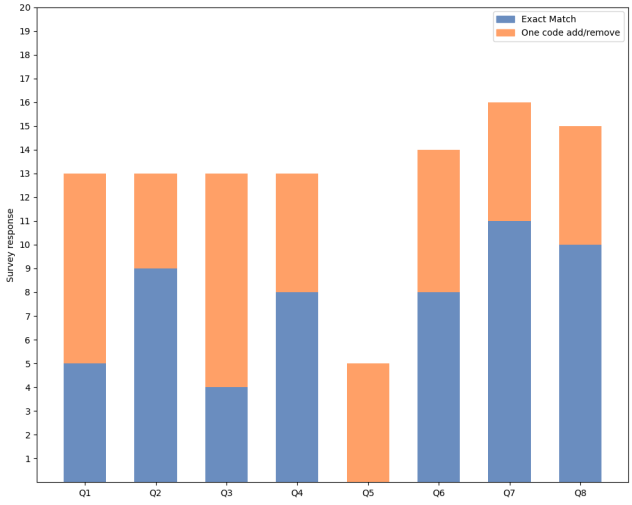

Figure 7. Distribution of exact matches and partial matches per survey question

generated codes that were not present in the ground truth. It is important to acknowledge that human annotators are also susceptible to errors, particularly when interpreting open-ended responses. To investigate further, we analyzed why the model added new codes or failed to generate certain expected codes. As discussed in **RQ$_3$**, although the model generated

at least one correct code for 47 out of 160 responses, these were not considered exact matches due to mismatches in the number or completeness of the codes, and potentially due to errors in the ground truth itself. For example, in the first row of Table 5, the model assigned the code BadAnswer, which was not present in the human annotations. According to the codebook, BadAnswer should be used when the response does not address the question or is irrelevant. In the corresponding survey response, the participant begins with "not sure if there is a way", which the model interpreted as a non-answer, justifying the use of the BadAnswer code. Based on the definition, this code appears to be appropriate and may have been mistakenly omitted by the human annotator. In the second example, the model assigned the code AutomatedTool, which again was not present in the ground truth. The code definition specifies that this label should be used when the response mentions the use of a tool. In the given response, the term "API" is mentioned, which the model interpreted as a reference to a tool, thereby assigning the code AutomatedTool. This assignment appears justifiable based on the definition. In contrast, the third response in the table demonstrates a failure case. The ground truth contains two codes: DetectBiases and Reproducibility. The model correctly assigned DetectBiases but failed to assign Reproducibility. According to the codebook, Reproducibility applies when the response discusses simplifying work or making processes easier. However, the survey response did not contain any reference to ease of work or process simplification, which likely explains why the model did not assign this code. In the Figure 7, for question 5 the model failed to assign code for

These findings suggest that some of the mismatches between model output and human annotation may not necessarily reflect model errors, but rather reveal the subjectivity and potential inconsistencies in human coding.

> **Summary of RQ$_4$:** This analysis highlights that some mismatches between model-generated and human-coded responses may stem from human oversight rather than model error. In several cases, the model correctly applied codes based on definitions, while human annotators may have missed them. These findings underscore the subjectivity and inconsistency that can occur in manual qualitative coding.

4.5 RQ$_5$: What is the impact of agents agreements on exact match?

As shown in Figure 6, our approach generated more codes for survey question Q7: `What main challenges do you foresee in the creation and use of DataBOMs?` To evaluate the consistency of code assignments, we used a

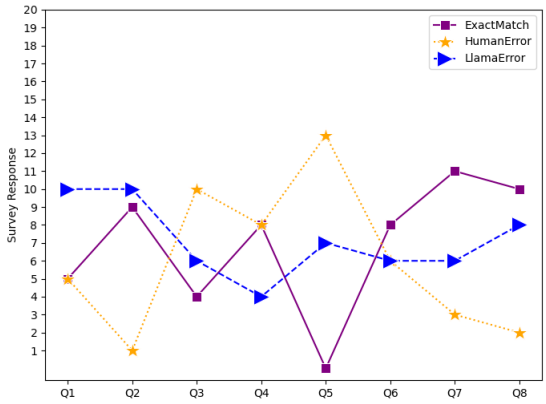

Figure 8. Comparison of exact matches, human annotation errors, and llama model errors across survey questions

3-agent, 3-prompt setup, resulting in a total of 9 decisions per survey response (Figure 2). Across Figures 6, 5, 7, and 8, we report results based on agent agreement. Specifically, if at least 6 out of 9 decisions returned "yes," we accepted the code for that response. However, to explore the effect of decision thresholds, we also varied the required agreement from 3 to 9 yes votes.

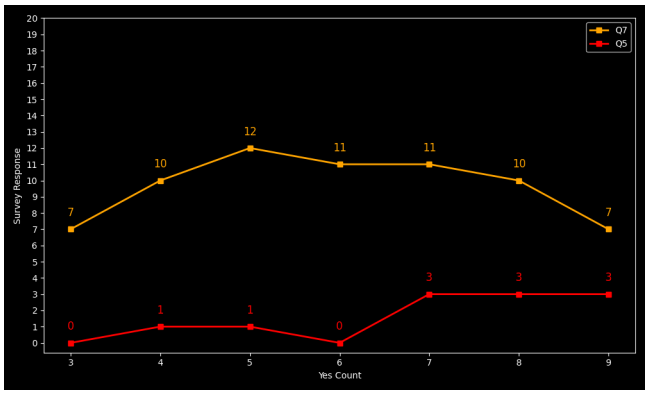

Figure 9. Exact match count across different levels of agreement in a 3-agent, 3-prompt LLM framework (total of 9 decisions per case)

Figure 9 illustrates how exact match performance changes with different yes count thresholds. When the threshold was set to 3, the model achieved 7 exact matches. With 5 yes votes, it peaked at 12 exact matches. Interestingly, increasing the threshold beyond this point led to a drop in exact match performance. At the strictest level 9 yes votes the model again produced only 7 exact matches, the same as with the lowest threshold. This suggests that requiring too much agreement may limit the model's ability to assign correct codes.

Table 5. Survey response, code and definition

No.	Survey response	Code definition	Code	Verdict
1	Not sure if there is a way, at the moment. An automated tool for checking that what is reported in the AIBOM is correct and complete would be useful. I do not know to what extent that would be feasible, though.	The response doesn't answer the asked question. The response is incoherent, rambling, etc.	BadAnswer	Added
2	Universal API can provide this information.	Response proposes the use of a tool, even if the specific details aren't clear.	AutomatedTool	Added
3	As with DataBOMs, one big benefit is the ability to better detect biases and reproduce results.	AIBOMs make it easier to reproduce a work.	Reproducability	Removed

Summary of RQ_5: We analyzed how varying the agreement threshold among 9 LLM decisions affected exact match performance. The results show that a moderate threshold (e.g., 5 or 6 yes votes) yields the highest number of exact matches, while stricter thresholds reduce the model's ability to assign correct codes.

5 THREATS TO VALIDITY

Internal Validity

The choice of data sources may influence our study. To ensure the internal validity of our study, we kept the experimental setup consistent throughout the evaluation of the LLMs. All survey responses were analyzed using the same prompts, a fixed set of codes, and identical model configurations across runs. We also manually reviewed cases to understand whether mistakes came from the model or possible issues in the human-coded data. To further improve reliability, we compared outputs from multiple agents instead of relying on a single model's decision.

External validity The external validity of our study is limited to the specific setting of software engineering survey data and the fixed codebook we used for coding. In our study, we achieved approximately 30% accuracy, highlighting the need for further refinement and validation of the approach. Since the prompts were designed for software engineering, the results might not transfer well to other domains. Future research using a wider range of datasets and coding approaches is needed to understand how well this method works in different contexts.

6 CONCLUSTIONS

In this study, we explored the effectiveness of large language models, particularly LLaMA 3 variants, in supporting qualitative open coding of survey responses in the context of software engineering. We implemented a multi-agent evaluation framework that involved three LLaMA models (LLaMA3-70B, LLaMA3-70B-Instruct, and LLaMA3-8B) and applied three structurally distinct prompts to assess model performance across nine combinations. Our findings reveal that while exact code matches between model outputs and human annotations were limited, the models were still able to produce partially aligned codes in a majority of cases. Through detailed error analysis, we identified frequent patterns of code addition and omission. Notably, the models often added new codes that were not present in the ground truth, raising questions about whether these were genuine model errors or potential oversights in human annotation. Manual inspection showed that, in some instances, model-generated codes were justifiable based on the code definitions, indicating the potential for LLMs to offer valuable second opinions or corrections in the coding process. While important, we also emphasized that exact match evaluation does not fully capture the nuanced capabilities of LLMs in qualitative analysis. As demonstrated in our results, even when the model failed to match the full set of human-assigned codes, it often identified at least one relevant code, suggesting potential for supporting or augmenting human coding efforts. Overall, this work highlights both the promise and limitations of LLMs in qualitative coding tasks. While current models are not yet reliable enough to replace human coders entirely, they can provide meaningful assistance and flag cases for deeper human review. Future work should explore fine-tuning LLMs on domain-specific annotation examples, incorporating human-in-the-loop feedback, and

improving prompt design to enhance model reliability and interpretability in qualitative research settings.

References

[1] Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training LLMs for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[2] Julia Bailey. 2008. First steps in qualitative data analysis: transcribing. *Family practice* 25, 2 (2008), 127–131.

[3] Amanda Barany, Nidhi Nasiar, Chelsea Porter, Andres Felipe Zambrano, Alexandra L Andres, Dara Bright, Mamta Shah, Xiner Liu, Sabrina Gao, Jiayi Zhang, et al. 2024. ChatGPT for education research: exploring the potential of large language models for qualitative codebook development. In *International conference on artificial intelligence in education*. Springer, 134–149.

[4] Youngjin Chae and Thomas Davidson. 2023. Large language models for text classification: From zero-shot learning to fine-tuning. *Open Science Foundation* (2023).

[5] Kerry Dhakal. 2022. NVivo. *Journal of the Medical Library Association: JMLA* 110, 2 (2022), 270.

[6] Tyna Eloundou, Sam Manning, Pamela Mishkin, and Daniel Rock. 2023. Gpts are gpts: An early look at the labor market impact potential of large language models. *arXiv preprint arXiv:2303.10130* (2023).

[7] Yunhe Feng, Sreecharan Vanam, Manasa Cherukupally, Weijian Zheng, Meikang Qiu, and Haihua Chen. 2023. Investigating Code Generation Performance of ChatGPT with Crowdsourcing Social Data. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. 876–885. doi:10.1109/COMPSAC57700.2023.00117

[8] Jie Gao, Yuchen Guo, Gionnieve Lim, Tianqin Zhang, Zheng Zhang, Toby Jia-Jun Li, and Simon Tangi Perrault. 2024. CollabCoder: A Lower-barrier, Rigorous Workflow for Inductive Collaborative Qualitative Analysis with Large Language Models. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 11, 29 pages. doi:10.1145/3613904.3642002

[9] Google. 2025. Gemini. https://gemini.google.com/app/. April 2025 version.

[10] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[11] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*. 933–944.

[12] Perttu Hämäläinen, Mikke Tavast, and Anton Kunnari. 2023. Evaluating Large Language Models in Generating Synthetic HCI Research Data: a Case Study. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 433, 19 pages. doi:10.1145/3544548.3580688

[13] Sungsoo Hwang. 2008. Utilizing qualitative data analysis software: A review of Atlas. ti. *Social Science Computer Review* 26, 4 (2008), 519–527.

[14] Md Robiul Islam. 2024. Application of Multimodal Large Language Models in Autonomous Driving. *arXiv preprint arXiv:2412.16410* (2024).

[15] Bernard J Jansen, Soon-gyo Jung, and Joni Salminen. 2023. Employing large language models in survey research. *Natural Language Processing Journal* 4 (2023), 100020.

[16] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. *arXiv preprint arXiv:2406.00515* (2024).

[17] Udo Kuckartz and Stefan Rädiker. 2019. *Analyzing qualitative data with MAXQDA*. Springer.

[18] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[19] Meta. 2025. Llama. https://www.llama.com/models/llama-3/. April 2025 version.

[20] Meta. 2025. Ollama. https://github.com/ollama/ollama. April 2025 version.

[21] Mohamed Nejjar, Luca Zacharias, Fabian Stiehle, and Ingo Weber. 2025. Llms for science: Usage for code generation and data analysis. *Journal of Software: Evolution and Process* 37, 1 (2025), e2723.

[22] OpenAI. 2025. ChatGPT. https://chat.openai.com/. April 2025 version.

[23] Replication Package. [n. d.]. https://github.com/robiul-islam-rubel/LLMs-for-close-coding-software-engineering.

[24] Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. 2021. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446* (2021).

[25] Zeeshan Rasheed, Muhammad Waseem, Aakash Ahmad, Kai-Kristian Kemell, Wang Xiaofeng, Anh Nguyen Duc, and Pekka Abrahamsson. 2024. Can large language models serve as data analysts? a multi-agent assisted approach for qualitative data analysis. *arXiv preprint arXiv:2402.01386* (2024).

[26] Zeeshan Rasheed, Muhammad Waseem, Malik Abdul Sami, Kai-Kristian Kemell, Aakash Ahmad, Anh Nguyen Duc, Kari Systä, and Pekka Abrahamsson. 2024. Autonomous agents in software development: A vision paper. In *International Conference on Agile Software Development*. Springer Nature Switzerland Cham, 15–23.

[27] Trevor Stalnaker, Nathan Wintersgill, Oscar Chaparro, Massimiliano Di Penta, Daniel M German, and Denys Poshyvanyk. 2024. BOMs Away! Inside the Minds of Stakeholders: A Comprehensive Study of Bills of Materials for Software Systems. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 44, 13 pages. doi:10.1145/3597503.3623347

[28] Robert H Tai, Lillian R Bentley, Xin Xia, Jason M Sitt, Sarah C Fankhauser, Ana M Chicas-Mosier, and BG Monteith. 2023. Use of large language models to aid analysis of textual data. (2023).

[29] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020).

[30] Karthik Valmeekam, Sarath Sreedharan, Matthew Marquez, Alberto Olmo, and Subbarao Kambhampati. 2023. On the planning abilities of large language models (a critical investigation with a proposed benchmark). *arXiv preprint arXiv:2302.06706* (2023).

[31] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1398–1409.

[32] Alexander Wuttke, Matthias Aßenmacher, Christopher Klamm, Max M Lang, Quirin Würschinger, and Frauke Kreuter. 2024. AI Conversational Interviewing: Transforming Surveys with LLMs as Adaptive Interviewers. *arXiv preprint arXiv:2410.01824* (2024).

[33] Ziang Xiao, Xingdi Yuan, Q Vera Liao, Rania Abdelghani, and Pierre-Yves Oudeyer. 2023. Supporting qualitative analysis with large language models: Combining codebook with GPT-3 for deductive coding. In *Companion proceedings of the 28th international conference on intelligent user interfaces*. 75–78.

[34] Ziang Xiao, Xingdi Yuan, Q. Vera Liao, Rania Abdelghani, and Pierre-Yves Oudeyer. 2023. Supporting Qualitative Analysis with Large Language Models: Combining Codebook with GPT-3 for Deductive Coding.

In *Companion Proceedings of the 28th International Conference on Intelligent User Interfaces* (Sydney, NSW, Australia) *(IUI '23 Companion)*. Association for Computing Machinery, New York, NY, USA, 75–78. doi:10.1145/3581754.3584136

[35] Chunyan Zhang, Junchao Wang, Qinglei Zhou, Ting Xu, Ke Tang, Hairen Gui, and Fudong Liu. 2022. A survey of automatic source code summarization. *Symmetry* 14, 3 (2022), 471.

[36] Hao Zhong and Zhendong Su. 2015. An empirical study on real bug fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 913–923.