How Effective are Large Language Models in Generating Software Specifications?

Danning Xie*‡, Byoungwoo Yoo†‡, Nan Jiang*, Mijung Kim†¶, Lin Tan*, Xiangyu Zhang*, Judy S. Lee§
*Purdue University, West Lafayette, IN, USA
{xie342, jiang719, lintan}@purdue.edu, xyzhang@cs.purdue.edu
†UNIST, Ulsan, South Korea
{captainnemo9292, mijungk}@unist.ac.kr
§ADP, Roseland, NJ, USA
judy.lee@adp.com

***Abstract*— Software specifications are essential for many Software Engineering (SE) tasks such as bug detection and test generation. Many existing approaches are proposed to extract the specifications defined in natural language form (e.g., comments) into formal machine-readable form (e.g., first-order logic). However, existing approaches suffer from limited generalizability and require manual efforts. The recent emergence of Large Language Models (LLMs), which have been successfully applied to numerous SE tasks, offers a promising avenue for automating this process. In this paper, we conduct the *first* empirical study to evaluate the capabilities of LLMs for generating software specifications from software comments or documentation. We evaluate LLMs' performance with Few-Shot Learning (FSL) and compare the performance of 13 state-of-the-art LLMs with traditional approaches on three public datasets. In addition, we conduct a comparative diagnosis of the failure cases from both LLMs and traditional methods, identifying their unique strengths and weaknesses. Our study offers valuable insights for future research to improve specification generation.**

Index Terms—**software specifications, large language models, few-shot learning**

I. Introduction

Accurate and comprehensive software specifications are essential for ensuring the correctness, dependability, and quality of software systems [1]–[4]. Common software specifications include pre- and post-conditions for a target function that describes the constraints of input parameters and the expected behaviors or output values. They are often required or crucial for generating effective test cases and test oracles, symbolic execution, and abnormal behavior identification [3], [5]–[7].

Numerous approaches have been proposed to advance automation in extracting specifications from software texts (e.g., documents or comments) into machine-readable forms, including rule-based methods [1], [3], [8], ML-based methods [4], [9], [10], search-based methods [11], etc. For example, Jdoctor [1] leverages pattern, lexical, and semantic matching to translate code comments into machine-readable specifications of pre-/post-conditions, which enables automated test generation that leads to fewer false alarms and the discovery of more defects. Several other attempts have been made to further improve these processes in various domains [9], [11]–[13]. However, most of existing work is domain-specific, relying on heuristics [1], [8], [10] or a large amount of manually annotated data [3], [11]. This reliance makes it challenging to generalize these approaches to other domains.

With the emergence of Large Language Models (LLMs), pre-trained on a tremendous amount of documents and source code [14]–[19], they have been applied to various Software Engineering (SE) tasks such as code generation [15], [20]–[25] program repair [26], [27], and reasoning [28]. These models have demonstrated competitive performance compared to traditional approaches [15]–[17], [26], [29]. Given that software specification extraction predominantly involves the analysis and extraction from software texts, such as comments or documents, and the translation of natural language into (semi-)formal specifications, two research questions naturally arise: **(1) Are LLMs effective in generating software specifications from software texts? (2) What are the inherent strengths and weaknesses of LLMs for software specification generation compared to traditional approaches?**

A. Our Study

To fill in the gap, we conduct the *first* empirical study to evaluate the capabilities of LLMs in generating software specifications, in comparison with traditional approaches. First, due to the scarcity of labeled data in software specification extraction, we leverage LLMs with *Few-Shot Learning* (FSL) [30], a technique that enables LLMs to generalize from a limited number of examples. Second, we explore the potential of combining LLMs and FSL by investigating different prompt construction strategies and assessing their effectiveness. Third, we conduct an in-depth comparative diagnosis of the failure cases from both LLMs and traditional approaches. This allows us to pinpoint their unique strengths and weaknesses, providing valuable insights to guide future research and improvement of LLM applications. Fourth, we conduct extensive experiments, involving 13 state-of-the-art LLMs, and evaluate their performance and cost-effectiveness to facilitate model

‡The first two authors contributed equally to this paper.
¶Corresponding author.

selection in software specification generation. Our study setup and findings are shown below:

- **FSL with random examples outperforms traditional methods:** To assess the performance of LLMs with FSL, we first collect three available datasets from the previous specification extraction research, which contain software documents and comments, as well as the corresponding ground-truth specifications. We start with a basic prompt construction method that *randomly* selects examples for FSL. We then compare the results with the state-of-the-art specification extraction techniques. Our Finding 1 reveals that *with 10 – 60 randomly selected examples, LLMs' results that are comparable to (2.1% lower) or better than (0.8 – 4.3% higher) the state-of-the-art specification extraction techniques.*
- **Advanced prompt strategy enlarges the performance gap:** To further explore the potential of LLMs with FSL for specification generation, we evaluate and compare different prompt construction strategies in terms of their impact on the performance of LLMs. Such prompt construction strategies include the above-mentioned random selection and a *semantics-based* selection strategy. Finding 2 shows that *with a more sophisticated prompt construction method, the performance gap between LLMs and traditional approaches is enlarged (to 1.9 – 10.5%).*
- **LLMs and traditional techniques exhibit unique strengths and weaknesses:** While LLMs outperform traditional methods overall, our analysis of failure cases reveals noticeable differences in their failure patterns – Finding 3: *traditional methods often produce empty outputs, whereas LLMs tend to generate incomplete or ill-formed specifications*. This variation in failures prompts us to analyze the distinct capabilities of LLMs and traditional methods. We hence conduct an in-depth comparative diagnosis of the failure cases from both ends and investigate their root causes. We identified several unique challenges for LLMs, such as *ineffective prompts and missing domain knowledge, which account for 75% of their unique failures. In contrast, traditional methods fail uniquely 90% due to insufficient or incorrect rules derived from limited datasets.* (Findings 4 – 5)
- **Open-sourced CodeLlama-13B and StarCoder2-15B are the most competitive models:** Lastly, given the vast spectrum of LLMs in terms of their open-source availability, costs, and model sizes, it becomes imperative to understand their capabilities. We perform rigorous experiments on 13 popular state-of-the-art LLMs, e.g., CodeLlama-13B, GPT-4, etc., varying in designs and sizes, and evaluate their performance and cost-effectiveness in generating software specifications. Remarkably, our Findings 6 – 9 show that *most LLMs achieve better or comparable performance compared to traditional techniques. CodeLlama-13B and StarCoder2-15B are the overall most competitive model among the 13 evaluated models for generating specifications, with high performance, open-sourced flexibility and long prompt support. Their strong performance makes commercial models (e.g., GPT-4) less desirable due to size and cost.*
- **Identifying areas for future enhancement:** These findings enable us to identify challenges for further improvement in LLM applications, i.e., *hybrid approaches*, that integrate LLMs and traditional methods, and *improving prompts effectiveness*.

B. Contributions

This paper makes the following key contributions:

1) We conduct the *first* empirical study comparing the effectiveness of LLMs and traditional methods in generating software specifications from comments or documents and find that LLMs with FSL achieve results that are comparable to (2.1% lower) or better than (0.8 – 4.3% higher) the traditional methods with only 10 – 60 randomly selected examples (Section V-A).
2) We evaluate the impact of different prompt construction strategies on the FSL performance and find that the advanced strategy can further enlarge the performance gap between the LLM approach and traditional methods (to 1.9 – 10.5%) (Section V-B).
3) We present a comprehensive failure diagnosis, highlighting *unique* strengths and weaknesses of both traditional methods and LLMs, guiding future research (Section VI).
4) We extensively experiment on 13 state-of-the-art LLMs, assessing their performance and cost-effectiveness (Section VII).
5) We discuss the future directions for LLMs in generating software specifications including hybrid approaches and enhanced prompt design (Section VIII).
6) We release the artifacts in [31].

II. Background

1) Software Specifications: Software specifications describe software functionalities, behaviors, and usage, including pre- and post-conditions for functions to ensure correct use. For example, TensorFlow's API `tf.nn.max_pool3d` requires the parameter `input` to be a "5-D Tensor", failing which leads to exceptions. Specifications are critical for tasks like test generation [1], [3], program analysis [11], bug detection [2], [10], [13], and code synthesis [32]. These are typically from documents or comments in natural language form, requiring extraction into formal machine-readable formats for downstream tasks.

Various approaches [4], [8], [10], [13] have been developed to extract such specifications, but they rely on domain-specific heuristics or labeled data, limiting their generalizability. For example, Jdoctor [1] uses manually crafted patterns with pre-/post-processing, requiring coding and domain expertise.

2) Large Language Models (LLMs) and Few-Shot Learning (FSL): LLMs, pre-trained on extensive corpora of natural language and code, acquire general knowledge through tasks like masked span and next-token prediction [18], [19], [33],

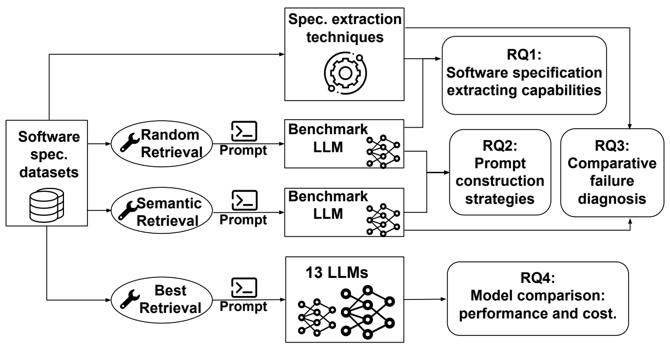

Spec. extraction techniques
RQ1: Software specification extracting capabilities
Software spec. datasets
Random Retrieval
Prompt
Benchmark LLM
RQ2: Prompt construction strategies
RQ3: Comparative failure diagnosis
Semantic Retrieval
Prompt
Benchmark LLM
Best Retrieval
Prompt
13 LLMs
RQ4: Model comparison: performance and cost.

Fig. 1: Study overview

[34]. To adapt pre-trained LLMs into customized tasks, fine-tuning [19], [26], [35], which requires significant labeled data, and prompting [18], which adapts LLMs using task-specific examples without modifying weights, are common approaches.

FSL enhances LLM performance on downstream tasks with limited labeled data [18], [30], [36]. In in-context FSL [18], [37], examples are provided in the input, allowing the model to generalize without altering weights. This makes prompting a practical and efficient method for specification generation.

III. Study setup

Fig. 1 presents an overview of our study. We collect available datasets from previous specification extraction work, containing software documents or comments and the corresponding ground-truth specifications. We then answer four research questions (RQs):

RQ1: How do LLMs with FSL perform compared to traditional rule-based approaches in extracting software specifications? We apply the benchmark LLM to the collected datasets and compare its performance with the traditional approaches. We use a basic prompt construction strategy, *random retrieval*, to construct prompts with random examples (Section III-B2a), which coach the LLMs to generate specifications by examples.

RQ2: How do prompt construction strategies affect the performance of the LLM approaches? We compare the performance of different prompt construction strategies, i.e., *Random Retrieval* and *Semantic Retrieval*. Semantic retrieval selects examples based on semantic similarity to the target context (Section III-B2b).

RQ3: What are the unique strengths and weaknesses of LLMs and traditional approaches? To provide better insights into the capabilities of different approaches and shed light on future research, we conduct a comparative failure diagnosis. In particular, we sample a set of cases that the LLM approach succeeds while the traditional approach fails and vice versa. We analyze the symptoms and root causes of these failures, and identify their unique strengths and limitations.

RQ4: How do different LLMs compare in terms of their performance and cost for generating software specifications? To assess performance and cost-effectiveness, we conduct extensive experiments with 13 state-of-the-art LLMs of different sizes, designs, and so forth. We employ the best-performing prompt construction strategy ("Best Retrieval" in Fig. 1) based on the results of RQ2.

TABLE I: Software specification datasets

Jdoctor-data [1]	Tag Type	@param	@return	@throws	Total
	#Annotations	243	139	472	854
DocTer-data [3]	Library	TensorFlow	PyTorch	MXNet	Total
	#Annotations	1,008	484	1,384	2,876
CallMeMaybe-data [10]	#Annotations		89		

Function signature	`isNullOrEmpty(java.lang.String string)`
Javadoc comment	`@return` true if the string is null or is an empty string
Specification	`string==null||string.isEmpty()→methodResultID==true`

Fig. 2: An example data point from Jdoctor-data.

A. Existing Specification Extraction and Data

We study three state-of-the-art rule-based approaches for specification extraction, Jdoctor [1], DocTer [3], and CallMeMaybe [10], as well as their respective datasets, containing annotated comments or documents with associated specifications. To avoid confusion, we use the terms Jdoctor-data, DocTer-data, and CallMeMaybe-data to refer to the datasets, while Jdoctor, DocTer, and CallMeMaybe refer to the three approaches. Table I presents the number of data points in each dataset. Rows "#Annotation" lists the number of document-specification pairs annotated

1) Jdoctor-data and Jdoctor: Jdoctor-data contains pre- and post-conditions written as executable Java expressions, translated from Javadoc comments of `@return`, `@param`, and `@throws` tags. Fig. 2 provides an example involving the post-condition for the function `isNullOrEmpty`. Jdoctor uses a combination of pattern, lexical, and semantic matching to identify key components, such as ⟨subject, predicate⟩ pairs, which are then converted into executable Java expressions through manually defined heuristics.

2) DocTer-data and DocTer: DocTer-data contains DL-specific specifications extracted from the API documents of TensorFlow, PyTorch, and MXNet. Specifications are categorized into four types: *dtype* (data types), *structure* (data structures), *shape* (parameter shape or number of dimensions), and *valid value* (valid ranges or enums). Fig. 3 shows an example data point. DocTer extracts constraints using syntactic rules constructed from annotated API descriptions

3) CallMeMaybe-data and CallMeMaybe: CallMeMaybe-data contains temporal constraints represented as event sequences, translated from natural language descriptions in code comments. Fig. 4 provides an example of a Java function, including its signature, comment, and the translated specifications that capture temporal relations between events `this.isEmpty()` and `this.clear()`. The extraction process of CallMeMaybe involves identifying propositions related to temporal dependencies and translating them into temporal constraints using semantic analysis and heuristic-based translations.

Function signature	`tf.image.extract_glimpse(input,size,offsets,...)`
Document description	input: A `` `Tensor` `` of type `` `float32` ``. A 4-D float tensor of shape `` `[batch_size,height,width,channels]` ``.
Specifications	***dtype***: `float32` ***structure***: `tensor` ***shape***: `[batch_size, height, width, channels]` ***ndim***: `4` ***range***: Null ***enum***: Null

Fig. 3: An example data point from DocTer-data.

Function signature	`clear()`
Comment	`Removes all of the elements from this priority queue. The queue will be empty after this call returns.`
Specification	`this.isEmpty() <- this.clear()`

Fig. 4: An example data point from CallMeMaybe-data.

Signature: <x_1 - signature>
Comment: <x_1 - comment>
Condition: <y_1>
...
Signature: <x_K - signature>
Comment: <x_K- comment>
Condition: <y_K>
Signature: <x_{target} - signature>
Comment: <x_{target} - comment>
Condition:

(a) Jdoctor/CallMeMaybe

Signature: <x_1 - signature>
Description: <x_1 - param.> - <x_1 - descp.>
Constraints: <y_1>
...
Signature: <x_K - signature>
Description: <x_K - param.> - <x_K - descp.>
Constraints: <y_K>
Signature: <x_{target} - signature>
Description: <x_{target} - param.> - <x_{target} - descp.>
Constraints:

(b) DocTer-data

Fig. 5: Prompt structures with target highlighted in orange.

Signature:	`min(float[] array)`
Javadoc:	`@param array` a nonempty array of float values
Annotation:	`(array.length==0)==false`
Generated:	`array.length>0`

Fig. 6: Semantically equivalent specs from Jdoctor-data

B. Specification Generation with LLMs

To extract specifications using an LLM, we construct prompts with examples, i.e., via few-shot learning.

1) Few-Shot Learning (FSL): Consider a dataset $D = (x_i, y_i)_{i=1}^{|D|}$, where x represents the *context* (e.g., document or comment) and y represents the target software specifications. For each data point (x_{target}, y_{target}), we select K examples from the other data points in D, excluding the target itself, using leave-one-out cross-validation [38]–[40]. These K examples, along with x_{target}, form the prompt used by the LLM to generate the output y_{out}, which is then compared to the ground-truth y_{target}.

2) FSL Prompt Construction: Fig. 5 shows simplified prompts for each dataset. For Jdoctor-data and CallMeMaybe-data, each of the K examples includes a function signature, comment, and the corresponding condition, with the target appended. For DocTer-data, the prompt includes a function signature, parameter description, and annotated constraints.

We treat the three tag types of Jdoctor-data (Table I) and the DocTer-data from three libraries as separate sub-datasets. For example, for a target of `@param` tag, we only select K examples from the other 242 data points of `@param` tag, excluding itself. Similarly, in DocTer-data, for a target, we select examples from the same library, excluding itself.

We study two commonly used strategies to choose the K examples: *random retrieval* and *semantic retrieval*.

a) Random Retrieval: It selects K examples randomly from the dataset, excluding the target, i.e., $D \setminus \{(x_{target}, y_{target})\}$. For instance, with K = 20, 20 random instances are selected as prompts, excluding the target.

b) Semantic Retrieval (SR): It selects examples semantically similar to the target, shown to be more effective than random retrieval [41], [42]. We use RoBERTa-large [43] due to its strong performance on the STS dataset [44]. The impact of different retrieval models is not studied, as previous research suggests minimal differences in performance [45].

3) Post-Processing of LLM Output: For Jdoctor-data, LLM completions can be semantically correct but not identical to the annotated specifications. Fig. 6 shows an example where both annotated (highlighted in yellow) and LLM-generated specifications (blue) convey the same condition but differ syntactically. Such equivalent but syntactically different specifications frequently occur in LLM results and makes automatic assessment challenging, especially when domain-specific knowledge is needed (e.g., array length is non-negative). Therefore, we manually inspect the generated completions for Jdoctor-data and report both the raw accuracy of perfect match and final accuracy after manual corrections. Two authors conducted independent reviews, resolving disagreements with a third.

Note that since Jdoctor uses a pattern-based method, it does not require such post-processing as the output format is constrained by heuristics. For DocTer-data and CallMeMaybe-data, we use perfect match, as post-processing is unnecessary for DocTer-data (orderless specifications) and CallMeMaybe-data (deterministic outputs).

C. Studied Large Language Models

Table II summarizes 13 LLMs from six series that we study in this paper, including the state-of-the-art generic, code-specific, open-sourced, and commercial LLMs. For open-source models, we focus on models that are smaller than 15B due to resource constraints. We do not focus on models with small sizes (e.g., 1B) as our preliminary experiments show their non-ideal performance. We run open-source models locally using 4 Nvidia RTX A6000 GPUs with 48GB memory.

D. Benchmark LLM

We choose to use CodeLlama-13B as the benchmark LLM for the study of RQs 1–3 (Fig. 1). CodeLlama-13B is one of the state-of-the-art open-source code LLMs, ensuring the reproducibility of our results. Additionally, it supports more input tokens (Table II), allowing a wide range of experimental settings, such as different numbers of examples in the prompt.

After identifying the best prompt construction strategies ("Best Retrieval" in Fig. 1) using CodeLlama-13B, we apply it to all other LLMs listed in Table II for the study of RQ4.

IV. Experimental Settings

A. Model Settings

As per our experimental design, we truncate examples from the beginning of the prompts to fit the token limit of each

TABLE II: Studied LLMs: sizes, token limits, prices per 1,000 tokens, and open-source statuses.

Model	#Param	Token limit	Price (per 1K)	Open-source?
GPT4 [46]	Unknown	8,192	$0.03	✗
GPT3.5 [47]	Unknown	16,384	$0.0015	✗
CodeLlama [48]	13B, 7B	16,384	-	✓
Llama3 [49]	8B	8,192	-	✓
Llama2 [50]	13B, 7B	4,096	-	✓
deepseek-coder [51]	6.7B	16,384	-	✓
StarCoder2 [52]	15B, 7B	16,384	-	✓
StarCoder [53]	15.5B	8,192	-	✓
CodeGen2 [54]	16B, 7B	2,048	-	✓

TABLE III: Comparison of CodeLlama-13B with random prompt construction and Jdoctor: Accuracy (%).

Approach	K	`@param`		`@return`		`@throws`		Overall	
		Raw	Processed	Raw	Processed	Raw	Processed	Raw	Processed
Jdoctor	-	97.0	97.0	69.0	69.0	79.0	79.0	83.0	83.0
CodeLlama-13B	10	81.1	89.7	35.3	48.9	76.3	**87.3**	71.0	81.7
CodeLlama-13B	20	84.4	92.6	39.6	55.4	73.9	88.0	71.3	**84.0**
CodeLlama-13B	40	92.2	94.2	48.2	61.2	79.9	90.3	78.2	86.7
CodeLlama-13B	60	91.8	94.7	48.9	56.8	83.3	92.4	80.1	87.3

model (as shown in Table II). If the median number of tokens in the prompts exceeds the limit, we skip that experiment. For instance, we skip the experiment for Jdoctor-data when $K = 60$ for CodeGen2 (with a token limit of 2,048) since the median number of tokens in the prompts is 3,737. To provide comprehensive results, we run the benchmark model (CodeLlama-13B) for RQ1 and RQ2 in all settings. All models' `temperatures` are set to 0 for minimal randomness.

B. Accuracy and F1 Metrics

To evaluate the correctness of the generated specifications for Jdoctor-data and CallMeMaybe-data, we use accuracy, defined as the ratio of correctly generated specifications to the total annotated specifications.

For DocTer-data, we follow the previous work [3] and use precision, recall, and F1 to evaluate the generated results for each specification category (e.g., *dtype*). For category t, let C_t be the number of correctly generated specifications, N_t be the total number of annotated specifications in the dataset, and G_t denote the number of generated specifications for category t. We define precision as $P_t = \frac{C_t}{G_t}$, recall as $R_t = \frac{C_t}{N_t}$, and F1 score as $F_t = 2 \cdot \frac{P_t \cdot R_t}{P_t + R_t}$. We report the overall precision, recall, and F1 across all four categories (*dtype*, *structure*, *shape*, and *valid value*) for each library (e.g., TensorFlow).

As discussed in Section III-B2, we treat Jdoctor-data of different tag types and DocTer-data of different libraries as separate datasets. We report the accuracy and F1 metrics for them separately, as well as the overall ones.

V. Evaluation Results

A. RQ1: Specification Extracting Capabilities

We evaluate LLMs on Jdoctor-data, DocTer-data, and CallMeMaybe-data using random retrieval strategy for prompt construction (Section III-B2a) with CodeLlama-13B as the subject model (Section III-D). Results for both CodeLlama-13B and baseline methods (Jdoctor, DocTer, and CallMeMaybe) are presented in Tables III, IV, and V. In the tables, we highlight (bold) the results of CodeLlama-13B that surpass the baseline methods with the fewest examples used.

TABLE IV: Comparison of CodeLlama-13B with random prompt construction and DocTer: Precision/Recall/F1 (%).

Approach	K	TensorFlow	PyTorch	MXNet	Overall
DocTer	-	90.0/74.8/81.7	78.4/77.4/77.9	87.9/82.4/85.1	85.4/78.2/81.6
CodeLlama-13B	10	68.3/74.3/71.2	72.9/69.1/70.9	66.4/71.8/69.0	68.2/72.2/70.1
CodeLlama-13B	20	77.9/72.5/75.1	76.6/72.2/74.3	71.7/69.7/70.7	74.7/71.1/72.8
CodeLlama-13B	40	77.8/80.4/79.1	78.7/72.7/75.5	75.3/75.2/75.2	76.7/76.6/76.6
CodeLlama-13B	60	80.9/79.8/80.4	79.6/76.4/**77.9**	78.3/80.4/79.4	79.4/79.5/79.5

TABLE V: Comparison of CodeLlama-13B with random prompt construction and CallMeMaybe: Accuracy (%).

Approach	K	Accuracy
CallMeMaybe	-	70.0
CodeLlama-13B	10	48.3
CodeLlama-13B	20	57.3
CodeLlama-13B	40	62.9
CodeLlama-13B	60	**70.8**

As described in Section III-B3, we manually post-process the specification generated for Jdoctor-data and present the raw accuracy (automatically calculated with perfect match) in col. "Raw" and the final accuracy (after manual correction) in col. "Processed" in Table III. For Jdoctor (row *Jdoctor*), these two values are the same. The columns "K" represent the number of examples in the prompts.

a) Results Summary: CodeLlama-13B demonstrates strong performance across all datasets. For Jdoctor-data (Table III), it achieves 84.0% accuracy using only 20 randomly chosen examples per comment type, surpassing Jdoctor (83.0%). It also outperforms Jdoctor for `@throws` and matches its performance for `@param` comments. For DocTer-data (Table IV), it reaches an F1 score of 79.5% with 60 examples per library, just 2.1% below DocTer (81.6%), despite DocTer's need for 2,696 annotated examples. For CallMeMaybe-data (Table V), CodeLlama-13B achieves 70.8% accuracy with 60 examples, outperforming CallMeMaybe by 0.8%.

Finding 1: CodeLlama-13B, with a small number (20 – 60) of randomly selected examples achieves comparable results (2.1% lower) with DocTer and outperforms the state-of-the-art specification extraction technique Jdoctor and CallMeMaybe by 0.8 – 4.3%.

B. RQ2: Prompt Construction Strategies

Tables VI, VII, and VIII reveal that CodeLlama-13B, when employing the SR strategy, outperforms all baseline methods (Jdoctor, DocTer, and CallMeMaybe), even with only 10 examples selected in the prompt from each type/category.

Fig. 7 showcases the effectiveness of random and SR strategies across prompt sizes. SR strategy consistently outperforms both the random strategy and traditional specification techniques across different prompt sizes, highlighting the importance of an appropriate prompt construction strategy for improved FSL performance.

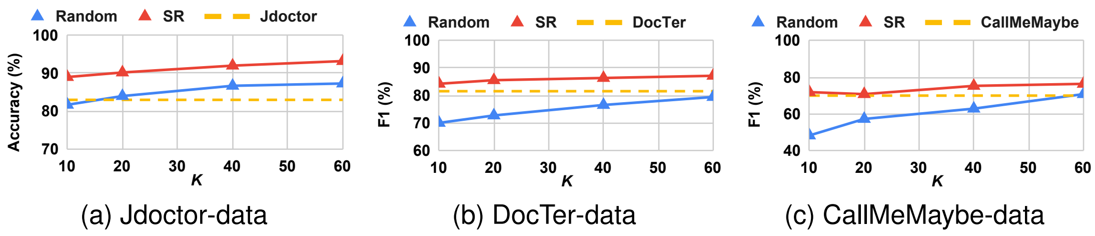


Fig. 7: Comparison of FSL performance using Random and Semantic Retrieval (SR) for prompt sizes (K) 10 – 60.

TABLE VI: Comparison of CodeLlama-13B using SR prompt construction and Jdoctor: Accuracy (%).

Approach	K	`@param`		`@return`		`@throws`		Overall	
		Raw	Processed	Raw	Processed	Raw	Processed	Raw	Processed
Jdoctor	-	97.0	97.0	69.0	69.0	79.0	79.0	83.0	83.0
CodeLlama-13B + **SR**	10	93.0	96.7	62.6	**71.2**	84.5	**90.3**	83.4	**89.0**
CodeLlama-13B + **SR**	20	94.7	**97.5**	63.3	73.4	86.4	91.3	85.0	90.2
CodeLlama-13B + **SR**	40	95.9	97.9	62.6	72.7	89.6	94.7	87.0	82.0
CodeLlama-13B + **SR**	60	95.9	98.8	65.5	75.5	90.3	96.0	87.9	93.5

TABLE VII: Comparison of CodeLlama-13B using SR prompt construction and DocTer: Precision/Recall/F1 (%).

Approach	K	TensorFlow	PyTorch	MXNet	Overall
DocTer	-	90.0/74.8/81.7	78.4/77.4/77.9	87.9/82.4/85.1	85.4/78.2/81.6
CodeLlama-13B + **SR**	10	82.8/82.4/**82.6**	81.6/80.2/**80.9**	86.0/87.4/**86.7**	84.1/84.4/**84.3**
CodeLlama-13B + **SR**	20	84.1/84.4/84.2	83.6/82.7/83.1	86.6/88.4/87.5	85.2/86.0/85.6
CodeLlama-13B + **SR**	40	85.9/85.8/85.9	83.4/83.4/83.4	87.5/88.4/87.9	86.2/86.6/86.4
CodeLlama-13B + **SR**	60	86.5/86.3/86.4	85.0/84.3/84.6	88.3/89.2/88.7	87.1/87.4/87.2

TABLE VIII: Comparison of CodeLlama-13B with SR prompt construction and CallMeMaybe: Accuracy (%).

Approach	K	Accuracy
CallMeMaybe	-	70.0
CodeLlama-13B + **SR**	10	**71.9**
CodeLlama-13B + **SR**	20	70.8
CodeLlama-13B + **SR**	40	75.3
CodeLlama-13B + **SR**	60	76.4

Finding 2: The semantic retrieval strategy further improves CodeLlama-13B's performance, leading to a 6.0 – 10.5% improvement over Jdoctor, a 2.7 – 5.6% increase over DocTer, and a 1.9 – 6.4% increase over CallMeMaybe.

VI. RQ3: Comparative Failure Diagnosis

In light of the outstanding performance of LLM compared with traditional techniques, it is crucial to delve deeper into their strengths and limitations. We manually examine failing cases of both LLM-based (e.g., CodeLlama-13B) and baseline approaches (i.e., Jdoctor, DocTer, and CallMeMaybe), and study their failure symptoms and root causes in a comparative manner, aiming to provide insights and directions for future techniques. Due to space constraints, we present results for the Jdoctor-data dataset, with the other two datasets available in the supplementary material [31]. The conclusions hold across all datasets, with no significant differences observed.

Fig. 8 presents the comparative performance of the CodeLlama-13B-based LLM method (L) and baseline method Jdoctor (J) as a Venn diagram. The number in the intersection ($L \cap J$) denotes cases where both methods are correct, while

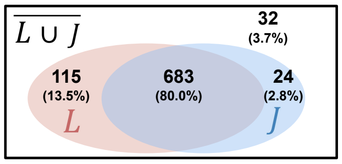


Fig. 8: Venn diagrams of specification generation. L: CodeLlama-13B; J: Jdoctor.

the number in section $\overline{L \cup J}$ indicates cases they both fail. The presented results are derived from the experiment using CodeLlama-13B with SR and $K = 60$ in RQ2 (Table VI).

Fig. 8 shows that both the LLM and Jdoctor perform well on the majority of cases (80.0%), indicating that the LLM quickly learns most specification extraction rules from a small number of examples in the prompts. The LLM has more (10.7%) unique correct cases than Jdoctor, indicating the generalizability of LLMs from extensive pretraining. There are a few cases where both methods fail, possibly due to inherent difficulties such as incomplete software text.

To better understand the pros and cons of different methods, we investigate the symptoms and the underlying causes of the failing cases. We *randomly* sample 30 cases from each section of Fig. 8 where at least one of the methods fails, i.e., $L \cap \overline{J}$, $\overline{L} \cap J$, and $\overline{L \cup J}$. For the sections that have less than 30 cases (e.g., $\overline{L} \cap J$), we sample all the failure cases.

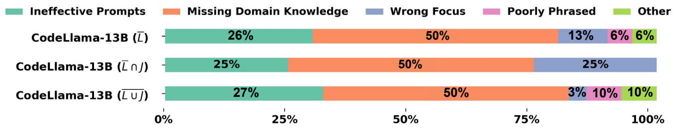
The sampling results in 147 failing cases of CodeLlama-13B and 135 of the baseline methods across three datasets, with a margin of error of 6% at a 90% confidence level. Two authors categorize these cases independently with a third author resolving disagreements.

A. Failure Symptom Analysis

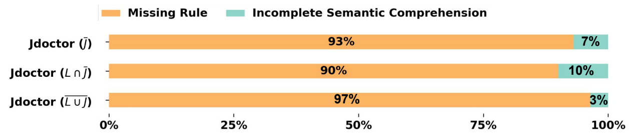
We conduct further analysis on the distributions of failure symptoms of both LLMs and traditional methods. The failure symptoms are classified into four categories, "ill-formed", "incorrect", "incomplete", and "empty". Category "*ill-formed*" refers to the generated specifications that are invalidly formed. "*Incorrect*" indicates specification errors, while "*Incomplete*" denotes the specification is a strict subset of the ground truth. "*Empty*" denotes a missing specification. The full results and more examples can be found in the artifacts [31].

We have the following observations:

a) LLMs are more likely to generate ill-formed and incomplete specifications: A small fraction (0–3%) of CodeLlama-13B's failures are ill-formed, unlike traditional



(a) Root causes of CodeLlama-13B



(b) Root causes of the baseline method (Jdoctor)

Fig. 9: Distributions of root causes on Jdoctor-data failures.

rule-based methods that guarantee valid outputs. In addition, LLMs are 8% more likely to produce incomplete specifications compared to rule-based methods, which reliably extract all types by directly matching sequences. In contrast, generative LLMs use sampling to decode outputs from a distribution, which may occasionally miss tokens.

b) Traditional techniques are much more likely to generate empty specifications than LLMs: For all three datasets, the most common failure of (rule-based) baseline methods is “Empty” (67%), where no specification is generated due to inapplicable rules. In contrast, LLMs generate results by predicting missing tokens and will only produce empty results when ”empty” is a valid outcome. Although with a lower empty rate, LLM tends to generate incomplete and ill-formed specifications as discussed in (a).

Finding 3: Compared to LLMs, traditional specification extraction approaches are much more likely to generate empty specifications, while LLMs are more likely to generate ill-formed or incomplete specifications.

B. Root Cause Analysis

In this section, we categorize the root causes of failures and study their distributions. At the end, we perform a comparative study based on the sections in the Venn diagrams (Fig. 8).

1) LLM Failure Root Causes: Since LLM results are difficult to interpret, it is in general difficult to determine the root causes of failing cases of LLMs. We employ the counterfactual method [55], [56]: a structured approach to identifying causal factors by altering one variable at a time to observe the impact on the outcome. In particular, we determine the root cause by finding a fix for it. The nature of the fix indicates the root cause. In some cases, the failure may be fixed in multiple ways. We consider the one requiring the least effort as the root cause and categorize them into five categories. Fig. 9a presents the distributions of the root causes of CodeLlama-13B in different sections of the Venn diagrams (Fig. 8). We now explain the five categories.

Ineffective Prompts: It means that the failure is due to the ineffectiveness of the examples in the prompt, even with SR. Although SR significantly improves FSL's performance

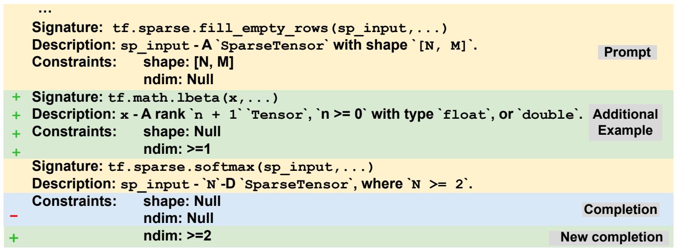

```
    ...
    Signature:  tf.sparse.fill_empty_rows(sp_input,...)
    Description: sp_input - A `SparseTensor` with shape `[N, M]`.               Prompt
    Constraints:     shape: [N, M]
                     ndim: Null
+   Signature: tf.math.lbeta(x,...)
+   Description: x - A rank `n + 1` `Tensor`, `n >= 0` with type `float`, or `double`.   Additional
+   Constraints:     shape: Null                                                        Example
+                    ndim: >=1
    Signature: tf.sparse.softmax(sp_input,...)
    Description: sp_input - `N`-D `SparseTensor`, where `N >= 2`.
    Constraints:     shape: Null                                               Completion
-                    ndim: Null
+                    ndim: >=2                                             New completion
```

Fig. 10: Example of “Ineffective Prompt”. Yellow, blue, and green denote the original prompt (simplified), generated completion, and an added example that enables LLM to generate the correct specification.

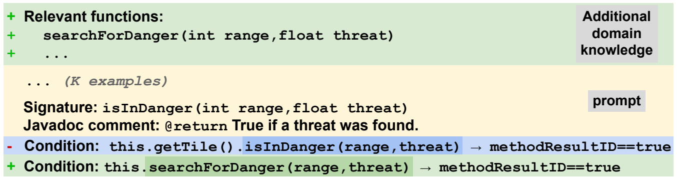

```
+ Relevant functions:                                                   Additional
+     searchForDanger(int range,float threat)                             domain
+     ...                                                               knowledge
  ... (K examples)
                                                                          prompt
  Signature: isInDanger(int range,float threat)
  Javadoc comment: @return True if a threat was found.
- Condition:  this.getTile().isInDanger(range,threat) → methodResultID==true
+ Condition: this.searchForDanger(range,threat) → methodResultID==true
```

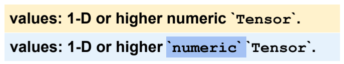
Fig. 11: Example of “Missing Domain Knowledge”. Yellow, blue, and green denote the original prompt (simplified), generated completion, added domain knowledge, and the new completion.

(Section V-B), it occasionally falls short in selecting the appropriate examples. If we can fix a failing case by manually selecting more relevant example(s) to the prompt, or simply altering the order of the examples in the original prompt, we consider the failure is due to ineffective prompts.

According to bar “CodeLlama-13B ($\bar{L}$)” from Fig. 9a, 26% CodeLlama-13B's failures on Jdoctor-data are due to this reason. We find that the order of examples plays a crucial role, as 21% of the failure cases in this category are resolved by rearranging the order of the examples.

Fig. 10 presents a portion of the prompt for the target parameter `sp_input`. CodeLlama-13B fails to generate the specification `ndim:>=2`, which is not explicitly stated in the description and requires CodeLlama-13B to comprehend the implicit relationship between `N` and its value range, i.e., “N$\geq$=2”. Adding an example with such implicit constraints enables CodeLlama-13B to generate the correct specification.

Missing Domain Knowledge: This refers to LLM's failures due to insufficient domain knowledge. For instance, in Fig. 11, CodeLlama-13B generates a specification using a non-existent function, `isInDanger`, instead of `searchForDanger`. This issue arises as the LLM lacks relevant context, such as the methods in the class, while Jdoctor employs a search-based approach examining all methods in the relevant classes. This result uncovers LLMs' limitation compared to traditional search-based methods: a deficiency in domain knowledge. To validate our hypothesis, we manually incorporate relevant domain knowledge into the prompt, alongside the provided examples. CodeLlama-13B then successfully generates the accurate specification, utilizing the correct function (in green). Fig. 9 shows that 50% of CodeLlama-13B's failures are due to missing domain knowledge.



values: 1-D or higher numeric `Tensor`.

values: 1-D or higher `numeric` `Tensor`.

Fig. 12: Example of "Wrong Focus". A minor adjustment (quoting keyword) enables generating correct specification.

Wrong Focus: It denotes instances where LLMs fail to focus on crucial keywords or are misguided or diverted by other content. For example, Fig. 12 shows the LLM fails to generate the correct specification from the original document (in yellow), `numeric`, which specifies the data type of the input tensor. By employing a slightly revised description that merely quotes the keyword, the LLM successfully generates the specification. Fig. 9 reveals that 13% of CodeLlama-13B's failures are due to the wrong focus.

To identify such failures, we apply three input mutation strategies: *minor modifications* by simply adding quotation marks to the keywords; *rewriting the sentence* while preserving the same syntactic structure (e.g., changing "A or B" to "B or A"); and *deleting redundant content* to help the LLM concentrate on the essential parts. All three strategies involve simple and semantics-preserving mutations and do not have impacts on the rule-based methods like DocTer as they rely on syntactic structure. We find that 42% of such cases can be resolved by merely quoting the keyword(s).

Poorly Phrased: It refers to instances where the original documents or comments are ambiguous, poorly written, or hard to understand. Rewriting the sentence to clarify its meaning enables LLMs to generate correct answers. According to Fig. 9, it contributs to 6% of CodeLlama-13B's failures.

Others: We group less common categories as "others", including "contradictory document" and "unclear", accounting for 6% of CodeLlama-13B's failures. The former refers to buggy or self-contradictory comments or documents, causing discrepancies between dataset annotations and LLM-generated specifications. "Unclear" indicates failures with unclear root causes, which we fail to fix despite various attempts.

Finding 4: The two dominant root causes combined (ineffective prompts and missing domain knowledge) result in 76% of CodeLlama-13B failures.

2) Baseline Methods Failure Root Causes: We manually investigated the sampled failing cases for Jdoctor, DocTer, and CallMeMaybe and identified three root causes: *missing rule*, *incomplete semantic comprehension*, and *incorrect rule*. The distributions on Jdoctor-data are in Fig. 9b.

Missing Rule: It refers to the absence of relevant rules or patterns, usually resulting in "empty" specifications (Section VI-A). A notable 93% baseline methods' failing cases fall into this category, exposing a limitation of rule-based methods: heavily dependent on manually defined or limited rules.

Incomplete Semantic Comprehension: This occurs when rule-based methods match part of a sentence but fail to grasp its full semantics, leading to incorrect results. For example, DocTer extracts a *structure* specification `vector` from "Initializer for the bias vector" but ignores full context or element relationships, affecting the correctness. This accounts for 7% of failures across the three datasets.

TABLE IX: Comparison of different LLMs with SR on Jdoctor-data: Accuracy (%) and Cost ($).

Approach/ Model (+SR)		K	`@param`		`@return`		`@throws`		Overall		Cost ($)
			Raw	Processed	Raw	Processed	Raw	Processed	Raw	Processed	
Jdoctor		-	97.0	97.0	69.0	69.0	79.0	79.0	83.0	83.0	-
GPT	4	60	90.1	95.5	63.3	77.7	90.0	94.9	85.7	**92.3**	117
	3.5	60	86.3	89.1	59.5	72.2	80.1	84.7	79.4	**84.4**	7.8
Codellama	13B	60	95.9	98.8	65.5	75.5	90.3	96.0	87.9	$\boxed{\mathbf{93.5}}$	-
	7B	60	95.5	98.4	60.4	72.7	90.9	95.3	87.2	**92.5**	-
deepseek-coder	6.7B	60	93.8	96.3	60.4	71.2	91.5	96.0	87.1	**92.0**	-
Llama3	8B	60	95.9	96.4	65.5	77.7	90.0	94.5	87.1	**92.9**	-
Llama2	13B	60	95.1	98.4	56.8	68.3	88.6	93.0	85.3	**90.5**	-
	7B	60	95.1	97.5	56.1	67.6	86.9	91.7	84.2	**89.4**	-
StarCoder2	15B	60	96.3	98.4	65.0	77.7	91.7	95.6	88.7	$\boxed{\mathbf{93.5}}$	-
	7B	60	95.9	97.9	64.0	76.3	90.5	94.5	87.7	**92.5**	-
StarCoder	16B	60	95.9	98.4	64.7	77.7	90.3	94.7	87.7	**93.0**	-
CodeGen2	16B	20	93.4	98.9	60.8	70.9	84.2	88.1	84.0	**89.0**	-
	7B	20	91.8	96.7	64.6	72.2	84.5	87.6	84.2	**88.3**	-

Incorrect Rules: This category denotes the cases where the applied rules are incorrect. This category of failure is unique to DocTer in the three tools we evaluated and is therefore not shown in the figure. DocTer automatically constructs the rules (map from syntactic patterns to specifications) based on their co-occurrence in the annotated dataset, which can potentially introduce incorrect rules, leading to incorrect extractions. 10% of DocTer's failing cases are due to *incorrect rules*, while Jdoctor and CallMeMaybe does not have any of such failures since their rules are all manually defined.

3) Comparative Root Cause Analysis: We compare the root cause distributions of CodeLlama-13B (Fig. 9a), with those of the baseline methods (Fig. 9b).

In cases where the baseline method succeeds (bar "CodeLlama-13B ($\overline{L} \cap J$)"), CodeLlama-13B's dominating failure causes are ineffective prompts, missing domain knowledge, and wrong focus. Notably, the wrong focus is particularly prevalent here, in contrast to section $\overline{L \cup J}$ where both approaches fail. For cases CodeLlama-13B succeeds (bar "Jdoctor ($L \cap \overline{J}$)") and the baseline method fails, we observe that the unique baseline failures are primarily due to missing rules and incomplete semantic comprehension. That is, when prompts and software texts are of high quality, *LLMs demonstrate outstanding generalizability*, unbounded by rule sets. They make predictions based on entire descriptions rather than partial ones. Notably, 10% of DocTer's unique failures are due to incorrect rules, all of which can be addressed by CodeLlama-13B. We suspect that any automatic rule inference techniques may suffer from such problems if human corrections are not in place.

Finding 5: Compared to traditional methods, CodeLlama-13B struggles with ineffective prompts and missing domain knowledge, causing 75% of its *unique* failures. LLMs, however, demonstrate excellent *generalizability*, whereas rule-based approaches often rely on insufficient or incorrect rules extracted from limited datasets.

TABLE X: Comparison of different LLMs with SR on DocTer-data: Precision/Recall/F1 (%), and Cost ($).

Approach / Model (+SR)		K	TensorFlow	PyTorch	MXNet	Overall	Cost ($)
DocTer		-	90.0/74.8/81.7	78.4/77.4/77.9	87.9/82.4/85.1	85.4/78.2/81.6	-
GPT	4	60	84.7/87.5/86.1	82.8/87.3/85.0	86.8/89.9/88.3	85.4/88.6/**87.0**	78
	3.5	20	81.9/83.7/82.8	77.3/81.2/79.2	84.3/87.3/85.8	82.3/85.0/**83.6**	5.2
CodeLlama	13B	60	86.5/86.3/86.4	85.0/84.3/84.6	88.3/89.2/88.7	87.1/87.4/**87.2**	-
	7B	60	86.4/83.6/85.0	84.4/80.4/82.4	88.2/88.8/88.5	86.9/85.6/**86.2**	-
deepseek-coder	6.7B	60	85.9/85.4/85.7	85.6/84.6/85.1	87.5/89.1/88.3	86.6/87.0/**86.9**	-
Llama3	8B	60	86.1/82.8/84.4	83.3/83.0/83.1	87.7/85.6/86.7	86.4/84.2/**85.3**	-
Llama2	13B	20	83.9/80.0/81.9	80.4/74.2/77.2	84.4/81.3/82.8	83.6/79.6/81.5	-
	7B	20	80.5/79.2/79.8	77.7/75.5/76.6	84.0/80.6/82.3	81.7/79.3/80.5	-
StarCoder2	15B	60	87.9/87.2/87.6	85.8/85.4/85.6	88.6/89.4/89.0	87.9/88.0/$\boxed{\textbf{87.9}}$	-
	7B	60	86.8/86.7/86.7	84.8/85.4/85.1	87.7/89.1/88.4	86.9/87.6/**87.2**	-
StarCoder	16B	60	85.0/86.7/85.8	83.4/84.6/84.0	87.4/89.2/88.3	85.9/87.5/**86.7**	-
CodeGen2	16B	10	79.7/81.4/80.6	77.6/79.3/78.5	85.0/86.3/85.7	81.9/83.4/**82.7**	-
	7B	10	77.7/77.3/77.5	76.6/77.2/76.9	82.2/84.0/83.1	79.7/80.5/80.1	-

TABLE XI: Comparison of different LLMs with SR on CallMeMaybe-data: Accuracy (%), and Cost ($).

Approach / Model (+SR)		K	Accuracy	Cost ($)
CallMeMaybe		-	70.0	-
GPT	4	60	**70.8**	15.15
	3.5	40	**73.0**	1.01
Codellama	13B	60	$\boxed{\textbf{76.4}}$	-
	7B	60	**75.3**	-
deepseek-coder	6.7B	60	66.3	-
Llama3	8B	60	**71.9**	-
Llama2	13B	40	**73.0**	-
	7B	40	69.7	-
StarCoder2	15B	60	$\boxed{\textbf{76.4}}$	-
	7B	60	**75.3**	-
StarCoder	16B	60	**73.0**	-
CodeGen2	16B	20	68.5	-
	7B	20	67.4	-

C. Generalizibility

To validate the generalizability of our analysis and conclusions, we extended our evaluation with two additional models, StarCoder-15.5B and GPT-3.5. The distributions from these models align with our findings and conclusions. Detailed results for these models are provided in the artifacts [31].

VII. RQ4: MODEL COMPARISON

Tables IX, X, and XI compare the performance and cost of 13 LLMs with the baseline methods (Jdoctor, DocTer, and CallMeMaybe). We only list the best results for each model with SR, and the full results can be found in the artifacts [31]. Model response time for generating specifications, subject to various factors such as environment, is omitted. Generally, the response time is reasonably quick for practical usage, ranging between 0.6 to 26.6 seconds. Due to the token limitation discussed in Section IV-A, some experiments are skipped. More experiments on DocTer-data are skipped since prompts for DocTer-data are much longer than those for other datasets, making them inapplicable for certain settings.

Overall, generic LLMs with as few as 10 domain examples achieve better or comparable performance as custom-built state-of-the-art specification extraction techniques such as DocTer. Specifically, 13, 10, and 9 out of the 13 models outperform traditional techniques Jdoctor, DocTer, and CallMeMaybe.

Finding 6: Most LLMs achieve better or comparable performance as custom-built traditional specification extraction techniques.

a) Best Performing Models: Among the 13 models, CodeLlama-13B and StarCoder2-15B achieve the best performance on all datasets, with an F1 score of 87.9% (Table X) and accuracies of 93.5% (Table IX) and 76.4% (Table XI).

Finding 7: CodeLlama-13B and StarCoder2-15B are the most competitive open-source models for extracting specifications, with among the highest performance, $0 cost, and long prompt support, facilitating its accessibility, adaptability, and customizibility.

b) Commercial Models (GPT-3.5 and GPT-4): GPT-3.5 and GPT-4 ($0.0015 and $0.03 per 1,000 tokens) achieve slightly worse performance than the best-performing models (e.g., CodeLlama-13B) given the same number of examples. Compared to CodeLlama-13B and StarCoder2-15B, which are free, open-source, and with much fewer parameters (Table II), commercial models (e.g., GPT-4) add no F1 or accuracy gains. Specifically, GPT-4's total cost of $32.8 (for $K = 10 - 60$) on CallMeMaybe-data causes a 5.6% accuracy degradation.

Despite the costs and risks of commercial models, such as charges for usage and concerns regarding the accessibility and continuity of research and applications, they are often more convenient, requiring only an API call with minimal hardware demands. In contrast, open-source models like CodeLlama-13B via Hugging Face APIs require more substantial hardware (e.g., GPUs) and technical expertise for configuration and optimization. Both options offer distinct advantages, allowing users to choose based on their needs, resources, and budget.

Finding 8: Commercial models (e.g., GPT-4) offer convenience and top-tier performance but with higher costs and risks such as accessibility, whereas open-source models are cost-effective and flexible alternatives but require greater technical expertise and hardware.

c) Other Open-Source Models (CodeLlama-7B, deepseek-coder-6.7B, Llama3, Llama2, StarCoder2-7B, StarCoder, and CodeGen2): StarCoder2-7B and CodeLlama-7B offer high performance, slightly below the best-performing model (by 0.7% – 1.7%), presenting it as a viable, smaller alternative to CodeLlama-13B and StarCoder2-15B. Conversely, Llama2, despite having the same model sizes as CodeLlama (7B and 13B), performs 3.0% – 5.7% worse, making it less suitable for this task. StarCoder-16B and Llama3 demonstrate their strong performance by outperforming the baseline methods by 1.9 – 10.0%. CodeGen2, on the other hand, has worse performance on the task. deepseek-coder-6.7B performs 3.7% worse than CallMeMaybe.

Finding 9: CodeLlama-13B and StarCoder2-15B yield the best performance on software specification generation among tested open-source models. Codellama-7B, StarCoder2-7B, StarCoder-16B, and LLama3-8B are reasonable open-source alternatives.

VIII. Challenges and Future Directions

Our analysis of failure cases highlights several challenges, pointing to future research directions in two areas:

a) Hybrid Approaches: Root cause analysis (e.g., Section VI-B3) indicates that combining the complementary strengths of LLMs and traditional methods could improve specification generation. Hybrid approaches can harness the generalizability of LLMs along with the domain-specific precision of traditional techniques to address gaps like missing domain knowledge. Promising work has begun in this area, integrating LLMs with software testing, program analysis [21], [22], [57], and retrieval-augmented generation [24], [25].

b) Improving Prompt Effectiveness: Improving prompts is another key direction to enhance LLM performance in specification extraction. Recent research on crafting more expressive, customizable, and domain-specific prompts [58]–[60] shows potential in guiding LLMs for better accuracy.

IX. Threats to Validity

a) Manual Evaluation: To address the equivalence specification issue in Jdoctor-data (Section III-B3), we manually evaluated results for RQ1, RQ2, and RQ4, and analyzed failure cases sampled in RQ3. To minimize biases, two authors *independently* conducted evaluations with 5.9% disagreement, resolving disagreements with a third author.

b) Analysis on Sampled Cases: In RQ3 (Section VI), we conduct a comparative analysis on a randomly sampled set of cases. This sampling approach could potentially limit the generalizability of our conclusions to the entire data population. To mitigate this concern, we extend the analysis to include failure cases from 2 additional models, GPT-3.5 and StarCoder (Section VI-C), where we observed consistent patterns, thereby reinforcing the robustness of our findings.

c) Data Leakage: Using public datasets introduces potential risks of data leakage. To address this, we analyze performance sensitivity to the number of examples in prompts (RQ1 and RQ2) and conduct zero-shot (ZSL) and one-shot (OSL) learning experiments. ZSL shows extremely poor performance (0–0.2% accuracy/F1), and OSL's performance is 28.9—77.2% lower than FSL, emphasizing the role of in-context learning.

X. related work

1) Software Specification Datasets and Extraction Methods: Traditional techniques for extracting software specifications from text, such as rule-based [1]–[4], [8], [13] or ML-based methods [1], [4], [12], shows limited generalizability across domains and require manual effort and domain knowledge. Our work is the first to study LLMs' capability on this task, leveraging FSL that offers improved generalizability and requires little annotated data. We evaluate LLMs on three datasets (Section III-A). Other than Jdoctor, techniques like @tcomment [8], Toradocu [2], and C2S [11] also extract specifications from Javadoc. They are excluded from this study as C2S is unavailable and the others are outdated or less effective [1], [11]. Advance [12] and DRONE [13] are excluded due to the absence of ground-truth specifications.

2) Large Language Models (LLMs): LLMs have been developed and used for a wide range of natural language understanding tasks such as question answering [18], [19], [35], [61]–[63] and natural language generating tasks such as text summarizing [18] and machine translation [18], [35], [62]. LLMs such as CodeLlama-13B have demonstrated their strong capabilities in numerous fields. We evaluate 13 state-of-the-art LLMs, varying in their design, sizes, etc., and discussed them in Section III. While other LLMs exist, they are not explored in this study as they are unsuitable for our task [19], less effective [17], [62]–[64], or unable to fit in our devices [65].

3) Applications of LLMs to SE tasks: LLMs have also been effectively applied to SE tasks such as code completion [15], [24], [25], [48], [53], test case generation [21], [22], [66], [67], program repair [27], [68], and software security [69]–[71], often outperforming traditional methods. However, LLMs have shown limitations in areas like code summarization [72], code suggestions [73], and software Q&A [74]. For example, developers frequently outperform LLMs in code-related tasks, highlighting the need for a comprehensive evaluation of LLMs in generating software specifications to identify their strengths and weaknesses in this specific SE context.

XI. Conclusion

We present the first empirical study that assesses the effectiveness of 13 LLMs for software specifications generation. Our findings reveal that most LLMs achieve better or comparable performance compared to traditional methods. Two of the best-performing models, CodeLlama-13B and StarCoder2-15B, as open-source models, outperform traditional approaches by 5.6 – 10.5% with semantically similar examples. Their strong performance makes closed-source commercial models (e.g., GPT4) less desirable due to size and cost. Additionally, we conduct a comprehensive failure diagnosis and identify the strengths and weaknesses of both traditional methods and LLMs. The two dominant limitations of LLMs are ineffective prompts and missing domain knowledge. Our study offers insights for future research to improve LLMs' performance on specification generation including hybrid approaches of combining traditional methods and LLMs, and improving prompts effectiveness.

XII. Acknowledgement

The authors thank the anonymous reviewers for their invaluable feedback and Guannan Wei for his input on the early draft. The research is partially supported by NSF 1901242 and 2006688 awards. It is also partially supported by the National Research Foundation grants funded by the Korean government (MSIT) (No. 2022R1A2C10911913 and 2022H1D3A2A0109297911), and the Institute of Information & Communications Technology Planning & Evaluation (IITP) grants funded by MSIT (No. RS-2023-00222830, RS-2024-00337414, and 2020-0-1336).

References

[1] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos, "Translating code comments to procedure specifications," *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018.

[2] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic generation of oracles for exceptional behaviors," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 213–224.

[3] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. W. Godfrey, "Docter: documentation-guided fuzzing for testing deep learning api functions," *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[4] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*icomment: bugs or bad comments?*/," in *Symposium on Operating Systems Principles*, 2007.

[5] E. Wong, L. Zhang, S. Wang, T. Liu, and L. Tan, "Dase: Document-assisted symbolic execution for improving automated software testing," *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 620–631, 2015.

[6] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on java predicates," *SIGSOFT Software Engineering Notes*, vol. 27, no. 4, pp. 123–133, 2002.

[7] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008, pp. 209–224.

[8] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tcomment: Testing javadoc comments to detect comment-code inconsistencies," *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 260–269, 2012.

[9] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language api descriptions," in *2012 34th international conference on software engineering (ICSE)*. IEEE, 2012, pp. 815–825.

[10] A. Blasi, A. Gorla, M. D. Ernst, and M. Pezzè, "Call me maybe: Using nlp to automatically generate unit test cases respecting temporal constraints," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–11.

[11] J. Zhai, Y. Shi, M. Pan, G. Zhou, Y. Liu, C. Fang, S. Ma, L. Tan, and X. Zhang, "C2s: translating natural language comments to formal program specifications," *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[12] T. Lv, R. Li, Y. Yang, K. Chen, X. Liao, X. Wang, P. Hu, and L. Xing, "Rtfm! automatic assumption discovery and verification derivation from library document for api misuse detection," *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[13] Y. Zhou, C. Wang, X. Yan, T. Chen, S. Panichella, and H. C. Gall, "Automatic detection and repair recommendation of directive defects in java api documentation," *IEEE Transactions on Software Engineering*, vol. 46, pp. 1004–1023, 2020.

[14] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[15] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A conversational paradigm for program synthesis," *arXiv preprint*, 2022.

[16] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," 2022. [Online]. Available: https://arxiv.org/abs/2204.05999

[17] W. Yue, W. Weishi, J. Shafiq, and C. H. Steven, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*, 2021.

[18] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *CoRR*, vol. abs/2005.14165, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[19] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[20] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *arXiv preprint arXiv:2305.01210*, 2023.

[21] Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng, "No more manual tests? evaluating and improving chatgpt for unit test generation," *arXiv preprint arXiv:2305.04207*, 2023.

[22] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests," *arXiv preprint arXiv:2207.10397*, 2022.

[23] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "Adaptive test generation using a large language model," *arXiv preprint arXiv:2302.06527*, 2023.

[24] Y. Ding, Z. Wang, W. Ahmad, H. Ding, M. Tan, N. Jain, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth *et al.*, "Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[25] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen, "Repocoder: Repository-level code completion through iterative retrieval and generation," *arXiv preprint arXiv:2303.12570*, 2023.

[26] N. Jiang, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023.

[27] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1592–1604.

[28] K. Pei, D. Bieber, K. Shi, C. Sutton, and P. Yin, "Can large language models reason about program invariants?" 2023.

[29] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Online: Association for Computational Linguistics, Jun. 2021, pp. 2655–2668. [Online]. Available: https://aclanthology.org/2021.naacl-main.211

[30] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. J. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *ArXiv*, vol. abs/2005.14165, 2020.

[31] "Artifacts," 2024. [Online]. Available: https://github.com/lt-asset/llm4spec

[32] R. David, L. Coniglio, M. Ceccato *et al.*, "Qsynth-a program synthesis based approach for binary code deobfuscation," in *BAR 2020 Workshop*, 2020.

[33] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in *Annual Meeting of the Association for Computational Linguistics*, 2019.

[34] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[35] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *CoRR*, vol. abs/1910.10683, 2019. [Online]. Available: http://arxiv.org/abs/1910.10683

[36] P. Riley, N. Constant, M. Guo, G. Kumar, D. Uthus, and Z. Parekh, "Textsettr: Few-shot text style extraction and tunable targeted restyling," 01 2021, pp. 3786–3800.

[37] S. M. Xie, A. Raghunathan, P. Liang, and T. Ma, "An explanation of in-context learning as implicit bayesian inference," in *International Conference on Learning Representations*, 2022. [Online]. Available: https://openreview.net/forum?id=RdJVFCHjUMI

[38] L. Breiman and P. Spector, "Submodel selection and evaluation in regression. the x-random case," *International statistical review/revue internationale de Statistique*, pp. 291–319, 1992.

[39] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, ser. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.

[40] M. Magnusson, M. Andersen, J. Jonasson, and A. Vehtari, "Bayesian leave-one-out cross-validation for large data," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 09–15 Jun 2019, pp. 4244–4253. [Online]. Available: https://proceedings.mlr.press/v97/magnusson19a.html

[41] O. Rubin, J. Herzig, and J. Berant, "Learning to retrieve prompts for in-context learning," *ArXiv*, vol. abs/2112.08633, 2021.

[42] J. Liu, D. Shen, Y. Zhang, B. Dolan, L. Carin, and W. Chen, "What makes good in-context examples for gpt-3?" *arXiv preprint arXiv:2101.06804*, 2021.

[43] "all-roberta-large-v1," https://huggingface.co/sentence-transformers/all-roberta-large-v1, Accessed: 2023.

[44] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019. [Online]. Available: http://arxiv.org/abs/1908.10084

[45] R. Shin, C. H. Lin, S. Thomson, C. C. Chen, S. Roy, E. A. Platanios, A. Pauls, D. Klein, J. Eisner, and B. V. Durme, "Constrained language models yield few-shot semantic parsers," *ArXiv*, vol. abs/2104.08768, 2021.

[46] OpenAI, "Gpt-4," 2024. [Online]. Available: https://platform.openai.com/docs/models/gpt-%204#gpt-4-turbo-and-gpt-4

[47] ——, "Gpt-3.5," 2024. [Online]. Available: https://platform.openai.com/docs/models/gpt-%204#gpt-3-5-turbo

[48] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[49] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[50] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[51] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[52] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.

[53] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[54] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, "Codegen2: Lessons for training llms on programming and natural languages," *arXiv preprint arXiv:2305.02309*, 2023.

[55] P. R. Rosenbaum, P. Rosenbaum, and Briskman, *Design of observational studies*. Springer, 2010, vol. 10.

[56] L. G. Neuberg, "Causality: models, reasoning, and inference, by judea pearl, cambridge university press, 2000," *Econometric Theory*, vol. 19, no. 4, pp. 675–685, 2003.

[57] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *International conference on software engineering (ICSE)*, 2023.

[58] S. Abukhalaf, M. Hamdaqa, and F. Khomh, "On codex prompt engineering for ocl generation: An empirical study," *arXiv preprint arXiv:2303.16244*, 2023.

[59] L. Beurer-Kellner, M. Fischer, and M. Vechev, "Prompting is programming: A query language for large language models," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 1946–1969, 2023.

[60] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 382–394.

[61] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019. [Online]. Available: http://arxiv.org/abs/1907.11692

[62] B. Wang and A. Komatsuzaki, "GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model," https://github.com/kingoflolz/mesh-transformer-jax, May 2021.

[63] S. Black, G. Leo, P. Wang, C. Leahy, and S. Biderman, "GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow," Mar. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5297715

[64] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[65] S. Black, S. R. Biderman, E. Hallahan, Q. G. Anthony, L. Gao, L. Golding, H. He, C. Leahy, K. McDonell, J. Phang, M. M. Pieler, U. S. Prashanth, S. Purohit, L. Reynolds, J. Tow, B. Wang, and S. Weinbach, "Gpt-neox-20b: An open-source autoregressive language model," *ArXiv*, vol. abs/2204.06745, 2022.

[66] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 423–435.

[67] S. Kang, J. Yoon, and S. Yoo, "Large language models are few-shot testers: Exploring llm-based general bug reproduction," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023.

[68] H. Ruan, Y. Zhang, and A. Roychoudhury, "Specrover: Code intent extraction via llms," *arXiv preprint arXiv:2408.02232*, 2024.

[69] X. Xu, Z. Zhang, Z. Su, Z. Huang, S. Feng, Y. Ye, N. Jiang, D. Xie, S. Cheng, L. Tan *et al.*, "Leveraging generative models to recover variable names from stripped binary," *arXiv preprint arXiv:2306.02546*, 2023.

[70] D. Xie, Z. Zhang, N. Jiang, X. Xu, L. Tan, and X. Zhang, "Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 4554–4568.

[71] Z. Su, X. Xu, Z. Huang, Z. Zhang, Y. Ye, J. Huang, and X. Zhang, "Codeart: Better code models by attention regularization when symbols are lacking," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3643752

[72] W. Sun, C. Fang, Y. You, Y. Miao, Y. Liu, Y. Li, G. Deng, S. Huang, Y. Chen, Q. Zhang *et al.*, "Automatic code summarization via chatgpt: How far are we?" *arXiv preprint arXiv:2305.12865*, 2023.

[73] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. J. Jiang, "Github copilot ai pair programmer: Asset or liability?" *Journal of Systems and Software*, vol. 203, p. 111734, 2023.

[74] B. Xu, T.-D. Nguyen, T. Le-Cong, T. Hoang, J. Liu, K. Kim, C. Gong, C. Niu, C. Wang, B. Le *et al.*, "Are we ready to embrace generative ai for software q&a?" *arXiv preprint arXiv:2307.09765*, 2023.