Current developments in Large Language Model based Debugging (2023, 2024)

Alexander Engelhardt
department of computer science
University of Helsinki
Helsinki, Finland
alexander.engelhardt@helsinki.fi

***Abstract*—In the last couple of years, we have seen Large Language Models (LLMs) enter the mainstream with applications in various domains. One such domain is the use of LLMs in debugging of software. This paper aims to map out how LLM-based debugging is developing right now (2023, 2024). To do this a semi-systematic literature review of the freshest studies is performed. The primary focus is on analyzing important theoretical concepts and architectural design decisions to identify lasting trends. The exact results of studies are a secondary focus as these are often difficult to validate and subject to being outdated as new studies roll out.**

The main contributions of this paper are three-fold. First, the field of LLM-based debugging is at an early stage. This is clear from the fact that there is little consensus on what techniques are effective with radically different approaches tested to achieve results. Second, this paper has identifies the key design decisions when creating an LLM-based debugging tool. These are 1. the specific LLM-used, 2. fine-tuning process, 3. prompt-engineering, 4. tool-integration, and 5. thought model. The third contribution of this paper regards the general strengths and weaknesses of LLM-based debugging. The big picture shows that LLMs are an excellent base technology for debugging tools, achieving *state-of-the-art or better* results. When faced with bugs of greater complexity the effectiveness seems to drop.

By reading this paper the reader should get a broad picture of the prospects and challenges related to LLM-based debugging. Being informed on the possibilities of LLM-based debugging hopefully spurs further research on the topic. Being aware of the challenges is important to avoid misunderstandings and common pitfalls.

***Index Terms*—software engineering, large language models, debugging, automated program repair**

I. Introduction

Though Large Language Models (LLMs) have been around for a while, recent advances have made generating code based on natural language a reality [1]. Since this advancement, a plethora of LLM-based tools, that can be used for software development, have emerged, such as ChatGPT and Github Copilot. Even more recently the fixing of broken code with help from LLMs - *LLM-based debugging* - has gathered a lot of interest. LLM-based debugging can be defined as any form of debugging where the usage of LLM is central to the process. This might entail direct fixing of broken code, or assistance in the form of explaining how to solve a bug. The debugging process is estimated to take up over half of the total time spent in software development and is thus a key area with a need for increased effectiveness [2].

To investigate the potential of LLM-based debugging, this paper reports the findings of a semi-systematic literature review of the most recent studies (2023, 2024). The motivation for this laser focus is that the recent advances in LLM technology could be argued to be a paradigm shift with a clear *before* and *after*. Additionally, it cuts down the number of studies to a degree where comparing studies on a more detailed scale is possible.

A foremost challenge for this paper relates to the validity of the current literature on LLM-based debugging. First, the novelty of LLM-based debugging solutions means the literature is quite limited, with several articles being yet-to-be peer-reviewed. Second, the results of LLM-based debugging are hard to validate. This is due to LLMs often being trained on black-boxed data, which makes it impossible to know if the debugging benchmarks that have been used for evaluation of LLM-based debugging tools, have been present in the LLM training data. This is called the *generalization vs. memorization* problem.

This paper tries to avoid this problem by putting the main focus on studying the underlying factors that affect the results, while specific results are a secondary concern. This way it is possible to map out the general picture of this rapidly evolving field.

We can now introduce the research questions that we are trying to answer in this paper.

1) **How does the general field look like for LLM-based debugging?**
2) **What are the key design decisions for LLM-based debugging?**
3) **What are the general strengths and weaknesses of LLM-based debugging?**

The main findings of the literature review are two-fold. First, it shows that the field of LLM-based debugging is at an early stage with no general consensus on what is effective. Instead the field is charactarized by radically different approaches being tried out. Second, the key design decisions when creating LLM-based debugging tools are 1. the specific LLM used, 2. the fine-tuning process, 3. prompt-engineering, and 4. tool-integration. The fifth design decision is a concept coined within this paper; *thought model*. The concept of a thought model is used to describe how LLM-based tools can be augmented by chaining the output from one instance of an

LLM as the input to another instance of an LLM. Building on this concept different structures of varying complexity can be constructed.

The layout of the paper is the following. Section 2. explains concepts important to the context. Section 3. dives into the methodology of the semi-structured literature review. Section 4. lays out the results of the study. In Section 5. the results are discussed and analyzed to answer the research questions. Section 6. reports on the validity of this paper. Section 7. concludes the paper.

II. Background

A. Large Language Models

Large Language Models (LLMs) are sophisticated deep learning models created through a training process that involves several key steps.

Initially, raw text-data is collected and *tokenized*. Tokenization involves splitting text into smaller units like words or subwords. LLMs are built using the *transformer architecture*, first presented by [3]. The transformer architecture consists of several neural network layers and enables the capturing of long-range dependencies in sequential data. Each layer consists of a number of *parameters* that together determine the output for any given input to the LLM. The largest LLMs today have up to billions of parameters.

Using the the transformer architecture LLMs are trained on the tokenized data in an unsupervised learning process called *pre-training* [4]. This involves learning general language patterns by guessing the next token or filling in gaps.

Following pre-training, LLMs can be fine-tuned on more specific tasks using supervised learning [4]. This involves training the model on task-specific data to adapt its parameters and optimize its performance for tasks like language translation or code generation.

The two most important factors for how good an LLM is going to be at its given task are the size of the raw text-data it was trained on and the number of *parameters* in the LLM [5].

Next, a list of some of the most important things to note about LLMs is presented, based on [6]. **1.** Results are nondeterministic and may be completely untrue. When an LLM produces seemingly correct answers that are actually incorrect it is called *hallucination* and is one of the foremost challenges with LLMs. **2.** LLMs can only emulate logic and do not possess true logic. For example if you ask ChatGPT-3.5 who Tom Cruise's mother is, it will answer correctly; Mary Lee Pfeiffer. If you ask who Mary Lee Pfeiffer's son is, it doesn't know or will hallucinate an answer. **3.** Even the creators of LLMs are uncertain of how the internals of LLMs work.

B. Prompt Engineering

Once an LLM has been created users can interact with it giving it inputs or so-called *prompts*. The way inputs are formulated has proven to produce vastly different results [7]. This has led to the new craft of *prompt engineering*, which concerns the construction of successful prompts. As an example, *Chain-of-Thought* (CoT) prompting is way to formulate input where we demand the LLM to ”think” in steps to reach a conclusion.

C. Thought Model

This paper coins the term *thought model* to describe how chained LLM input-output flows can be structured using different models in an effort to achieve better results. More precisely the thought model describes how multiple instances of LLMs can cooperate.

The concept of a thought model is best explained through a set of examples presented in Fig. 1. Here we can see four different thought models labeled a.), b.), c.), and d.). The first thought model a.) ”input-output” is the simplest thought model where a LLM generates an output directly from an input. In this thought model there is only a single instance of a LLM present.

The second example b.) in Fig. 1. uses a more complex prompting strategy, but still uses the same basic thought model since there is still only one instance of an LLM present.

The third example c.) uses another thought model in the sense that it uses multiple parallel LLM instances to generates alternative outputs. In this specific case, the parallel thought model is further enhanced with the concept of *self-consistency*. Self-consistency entails comparing the alternative outputs and presenting the most prevalent output as the final output. That is, in self-consistency higher frequency among alternative outputs is regarded as being more likely to be the correct answer.

The fourth example d.), Tree of Thoughts, is the core idea presented in [8]. The Tree of Thoughts involves giving an LLM the ability to divide a task (input) into a tree-like structure. The idea is that this allows the LLM to create sub-tasks, explore different potential solutions, and backtrack without human interaction.

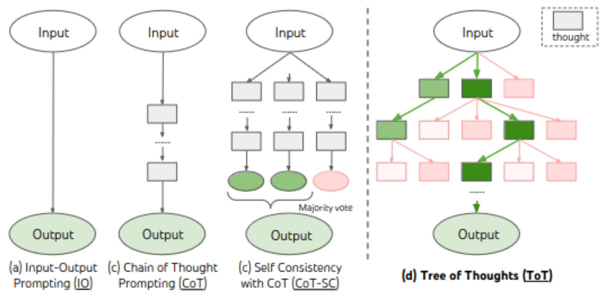

Input
Input
Input
Input
thought
Majority vote
Output
Output
Output
Output
(a) Input-Output Prompting (IO)
(c) Chain of Thought Prompting (CoT)
(c) Self Consistency with CoT (CoT-SC)
(d) Tree of Thoughts (ToT)

Fig. 1. Four different thought models for interacting with a LLM [8].

An important aspect of the concept of thought models is that they are able to be combined on a theoretical level. For example, we could construct a Tree of Self-consistent Thoughts where we use a tree structure for sub-task division, exploration and backtracking while each node in the tree consists of parallel LLMs finding the best solution through self-consistency.

III. Methodology

A. Semi-systematic Literature review

The methodology of this paper is a semi-systematic literature review with a gradual shift from being very structured to being more subjectively selective. The key steps can be viewed in Fig. 2. and are 1. Compose a search string, 2. Use search string on Google Scholar, 3. Use a set of criteria to filter articles based on the title and abstract, 4. Read the ”important parts” of the articles, 5. Do a second read through, taking standardized notes, 6. Analyze notes to produce results.

Since this is a fast-moving field with several back-to-back breakthroughs in the last couple of years this paper focuses on only the most recent literature. The definition for ”current” in this paper is thus the years 2023 and 2024.

Due to focusing on the most recent literature this literature review includes pre-prints. This way we do not exclude potentially important studies. To combat the concerns of validity, the focus is on higher-level design decisions and cross-study similarities, above specific results.

To find articles this paper utilizes Google Scholar. The idea behind using Google Scholar is that Google Scholar has less strict requirements which means also yet-to-be peer-reviewed articles are included.

B. Search string

To gather articles systematically we compose a search string. The key words that are used to find articles are *debugging* and *Large Language Models*. To widen the search we also include related terms or synonyms in the search string. For debugging these are *program repair* and *patching*. For Large Language Model we add *LLM, LLMs* and *LLM-based*. Additionally, alternative ways of spelling are also included in the search string.

An additional filter that is added is the requirement for the keywords to be found in the title of an article. This way we cut down the number of articles to those that have a specific focus on *LLMs and debugging*. Examples of the type of articles we want to exclude are articles on *LLMs and software development* where debugging might be mentioned but where it is not the main focus.

With the above-mentioned criteria we compose this search string: *allintitle:((”Debugging” OR ”Program-repair” OR ”Program repair” OR ”Patching”) (”LLM” OR ”LLMs” OR ”Large Language Models” OR ”LLM-based”))*

This string captures the focus of this paper in that it catches articles with a specific focus on the intersection of ”LLMs” and ”Debugging” without exclusion due to semantics.

At this stage, the search returns 27 articles on Google Scholar (date: 5.3.2024). All of these articles were reviewed based on their title and abstracts. Based on this, the articles were filtered according to certain criteria.

C. Exclusion criteria

1. The first criterion is that the article is about how LLM technology can be used for debugging code. For example, there are a few articles that present benchmarks that can be used to evaluate LLM-based debugging tools. These are excluded.

2. The second criterion is that the articles should focus on general debugging and not on the debugging of some specific/niche technology. For example, we remove an article that looks specifically at how LLMs can be used for debugging embedded systems.

3. The third criterion is that articles need to reach a certain level of base credibility. We include peer-reviewed articles, pre-prints, doctoral-, and master’s theses. One bachelor’s thesis is removed based on this criteria.

4. The fourth and last criterion is to remove duplicates or articles that were not available. To view the 27 articles and whether they were included or excluded see Appendix A1. See Table 1. for an overview of excluded articles. This table lists the reason for exclusion and the number of articles that were excluded based on this criterion.

Exclusion criterion	Number of articles
1. Off-topic	5
2. Not focused on general debugging	7
3. Below base level of credibility	1
4. Duplicate or not-available	3

TABLE I
EXCLUSION CRITERIA AND NUMBER OF ARTICLES EXCLUDED BASED ON THAT CRITERION.

After this step, there were 10 articles left. These articles were reviewed based on reading ”the important parts”, usually consisting of at least the introduction, the results, and the conclusion. The idea behind this general read-through was to get an overview of the articles. At this step, there was also the option to exclude articles based on some new found reason. One article was removed at this stage due to giving an unfocused impression, presenting two studies in a paper of only 9 pages. Additionally, it didn’t report validity nor was any of the tools publicly available.

D. Categories for notes

For the second read-through of the articles, standardized notes in a set of categories were taken. With this approach we have a systematic way of comparing articles. Using standardized categories is also necessary when analyzing the big picture of LLM-based debugging, as it allows for storing details which would be impossible to remember for the number of articles in question. The attention to detail helps with forming a true big picture and is more reliable than trusting memory and intuition.

The categories and their definitions are presented in Table 2 for a quick overview. Category 1. ”Article title” is self-explanatory. Category 2. ”Theme” signifies the main theme of the article. Category 3. ”Contribution” signifies the main contribution of the article. In this case, it is either the name of the tool (if article didn’t name the tool; a name was given) or an explanation of what was done. For categories 4. ”Thought model” and 5. ”Prompt-engineering”, see Chapter 3. for an explanation of what is meant with these concepts.

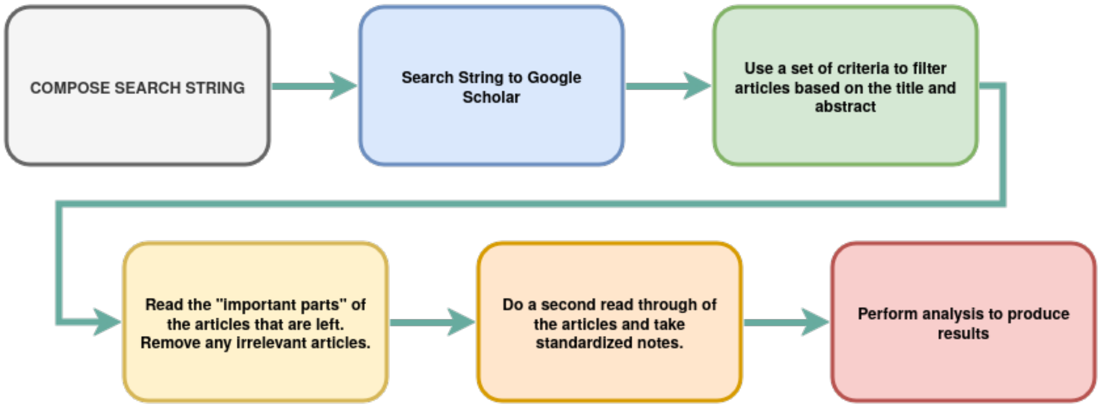

COMPOSE SEARCH STRING
Search String to Google Scholar
Use a set of criteria to filter articles based on the title and abstract
Read the "important parts" of the articles that are left. Remove any irrelevant articles.
Do a second read through of the articles and take standardized notes.
Perform analysis to produce results

Fig. 2. A birds eye view of the methodology of this semi-structured literature review.

Category 6. ”Results” is divided into two sub-categories; ”Results on easy to medium bugs” and ”Results on hard bugs”. Both sub-categories paraphrase remarks about the results of that particular study. The idea behind paraphrasing is that specific numbers are hard to validate and are not comparable. Thus it makes sense to paraphrase to get a more intuitive sense of the results while keeping in mind not to take these too seriously.

Category 7. ”Testing benchmark” signifies what benchmark was used to evaluate the debugging tool presented by the article. Category 8. ”LLM used” signifies the specific LLM/LLMs used by the debugging tool presented in the article. Category 9. ”Public” reports whether the debugging tool presented in the article is publicly available or not. Category 10. ”Validity” signifies whether the validity of the results was specifically discussed in the article or not. Category 11. ”Entity” signifies whether the article was written by someone from academia, industry, somewhere else, or a combination.

Category	Explanation
1. Article	The title of an article
2. Theme	The main theme of an article
3. Contribution	The main contribution of an article. E.g., the name of a tool or a key finding
4. Thought model	The architectural design of the LLM-based debugging tool
5. Prompt-engineering	The sort of prompt-engineering applied by the debugging tool presented in the article
6. Results	Paraphrased quote from the article
7. Testing benchmark	Against what testing benchmark was the LLM-based debugging tool evaluated
8. LLM-used	The specific LLM/LLMs used in the LLM-based debugging tool
9. Public	1=publicly available, 0=not available
10. Validity	1=validity reported, 0=not reported, explanation=something in between
11. Entity	Industry / Academia / Other / Mix /

TABLE II
DEFINING THE CATEGORIES FOR TAKING NOTES OF THE CHOSEN ARTICLES.

After taking standardized notes on every article the notes were reviewed and checked several times for correctness. After this, the notes were analyzed and synthesized to produce the main findings of this paper.

IV. Results

The results of this semi-structural literature review are achieved by analyzing standardized notes on all articles while adding additional information for a broader context. The subsections of this chapter follow the categories of the standardized notes, but closely related categories are combined, however. To make reading easier Fig 3. gives each article an ID and a nickname that will be used when referring to the articles. The nickname is either a keyword from the title or the name of the specific tool that was presented in the article.

A. *Themes and Contributions*

When looking at the general themes of the articles it can be stated that the current focus of the literature is on building LLM-based debugging tools and testing them. Each article’s main theme and contribution can be seen in Fig 4.

All but two articles present a specific concrete tool. These were Article 3. (LLM APR Era) [9] and Article 5. (Fine-tuning) [10]. Article 3. (LLM APR Era) focuses on comparing different LLMs and their debugging capabilities. Article 5 (Fine-tuning) compares how fine-tuning affects debugging as opposed to other factors. That is, the focus of these articles is not on creating some specific tool with its distinctive architecture. They focus instead on comparing the power of LLMs themselves or how these can be improved through fine-tuning while leaving out other factors.

The tools found in the articles vary in size from experiments to large-scale frameworks which are already being used to some extent in industry. For example, Article 9 (Inferfix) [11] presents a large-scale framework for end-to-end program repair using LLM technology. This framework is already being used internally at Microsoft to some extent. Most articles have not however reported their tools to be in active use at the current moment.

One of the key differences in approach is whether to solve bugs by emulating human debugging methods or by using a non-human approach. For example, Article 1 (RING) [12] conceptualizes program repair as *localization, transformation* and *candidate ranking*, resembling a human approach. Another article, Article 8. (Print-debugging) [13] also uses a very

Id	Title	Nickname
1	**Repair is nearly generation: Multilingual program repair with llms**	RING
2	**Copiloting the copilots: Fusing large language models with completion engines for automated program repair**	Copiloting
3	**Automated program repair in the era of large pre-trained language models**	LLM APR Era
4	**Better patching using LLM prompting, via Self-Consistency**	Self-consistency
5	**An empirical study on fine-tuning large language models of code for automated program repair**	Fine-tuning
6	**A Novel Approach for Automatic Program Repair using Round-Trip Translation with Large Language Models**	Round-trip-translation
7	**CigaR: Cost-efficient Program Repair with LLMs**	CigaR
8	**Leveraging print debugging to improve code generation in large language models**	Print-debugging
9	**Inferfix: End-to-end program repair with llms**	Inferfix

Fig. 3. Each article is given an id and a nickname to help with referring to them.

human approach by applying so-called print debugging. This involves giving the LLM the ability to run code, access console output, and insert print statements into key places to help it understand the execution flow.

Article 6 (Round-trip-translation) [14] leverages the power of LLM technology in an entirely different way from how a human would approach debugging. The idea is that bugs are fixed by first translating the faulty code into another language (programming or even natural) and then translating it back to the original language, fixing the code. This approach was inspired by the notion that grammatical mistakes can be fixed by applying round-trip translation.

Article 2 (Copiloting) [15] represents an approach where LLMs are given access to tools to improve performance. More specifically this article presents a tool that uses a completion engine to guide an LLM in the process of debugging. Article 4 (Self-consistency) [16] presents a tool that examines how LLM-based debugging can be improved by applying the thought model called self-consistency. Article 7 (CigaR) [17] puts special focus on the cost-efficiency of LLM-based debugging. This is the only article found within this literature review that has the cost-efficiency of LLM-based debugging as its prime focus.

Id	Nickname	Theme	Contribution
1	RING	LLMs vs. language specifc tools	RING (tool)
2	Copiloting	LLMs + Completion engines	Repilot (tool)
3	LLM APR Era	Comparing different LLMs	Larger models generally perform better
4	Self-consistency	Self-consistency	*SC-LLM-debugger* (tool)
5	Fine-tuning	Fine-tuning paradigm	Fine-tuning strategies
6	Round-trip-translation	Round-trip-translation	*RTT-debugger* (tool)
7	CigaR	Cost-efficiency	CigaR (tool)
8	Print-debugging	Print debugging	*PrintLLM* (tool)
9	Inferfix	End-to-end: A big framework	InferFix (tool), InferBugs (benchmark)

Fig. 4. The main themes and contributions of the articles.

B. Thought Model

Since the concept of a *Thought model* has been coined within this paper it is hard to extract direct references on this from the various articles. That said, the tools that have been found in the literature have been analyzed with the concept of thought model in mind, and in most cases, the thought model has been directly described, if not referred to as "thought model". To view the big picture of the different thought models used in the different articles view Fig 5.

Between the tools found in the literature, there are three main camps with variations within. These three camps are the simple thought model, the parallel thought model, and an augmented iterative thought model.

The articles where the simple thought model is present are Article 3 (LLM APR Era) and Article 5 (Fine-tuning). The key focus of these articles are on the LLMs themselves and thus it makes sense for them not to augment the results by other means.

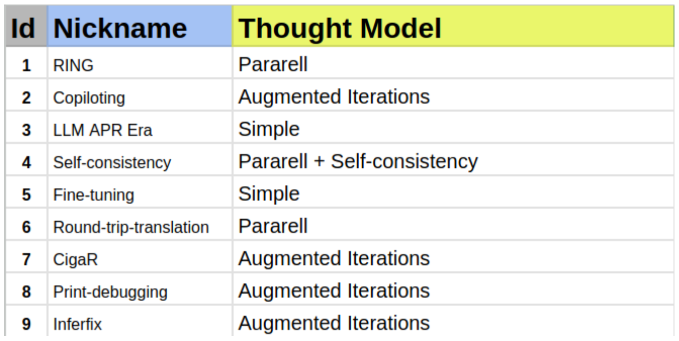
The parallel thought model is found in articles 1 (RING), 4 (Self-consistency), and 6 (Round-trip-translation). Article 1 (RING) applies a straightforward parallel approach. Article 6 (Round-trip-translation) doesn't apply a pure parallel approach. Instead, it generates one answer at a time and stops if a correct solution is found. This could be viewed as iterative, but since the input isn't altered between iterations it is effectively closer to a parallel thought model. Article 4 (Self-consistency) uses a parallel thought model with the addition of self-consistency.

The tools that use an augmented iterative thought model are articles 2 (Copiloting), 7 (CigaR), 8 (Print-debugging), and 9 (Inferfix). The augmentation is different in each case. Article 2 (Copiloting) augments each iteration with the updates from the completion engine. Article 8 (Print-debugging) does the same while entering a "debug process" between iterations. This debug process consists of inserting print statements at strategic points, executing the code, and analyzing the output.

No tool found within the literature uses a thought model like the Tree of Thought described by [8].

C. Prompting

When it comes to prompt engineering, several different strategies are applied. Each tool uses slightly different prompts, but there are also similarities between them. Some of the reoccurring strategies are *n-shot learning*, *cloze style*,



Id	Nickname	Thought Model
1	RING	Pararell
2	Copiloting	Augmented Iterations
3	LLM APR Era	Simple
4	Self-consistency	Pararell + Self-consistency
5	Fine-tuning	Simple
6	Round-trip-translation	Pararell
7	CigaR	Augmented Iterations
8	Print-debugging	Augmented Iterations
9	Inferfix	Augmented Iterations

Fig. 5. The thought models used in the LLM-based debugging tools.

and *chain-of-thought*. To view the big picture of the different prompting strategies see Fig 6.

N-shot learning is a prompting strategy where input is structured as follows:

input =

$$q_1 - a_1, q_2 - a_2, \ldots, q_n - a_n, Q - A_{blank} \text{ where:}$$

- q_i represents an example-question.
- a_i represents the corresponding example-answer.
- Q represents the question at hand.
- A_{blank} represents the answer we are requesting from the LLM.

The n-shot prompting strategy enables the LLM to learn from a set of example questions and answers (n-shots) and then apply this learned knowledge to generate the answer to the presented question Q.

N-shot learning has been applied in Article 1 (RING)and Article 3 (”APR LLM ERA”).

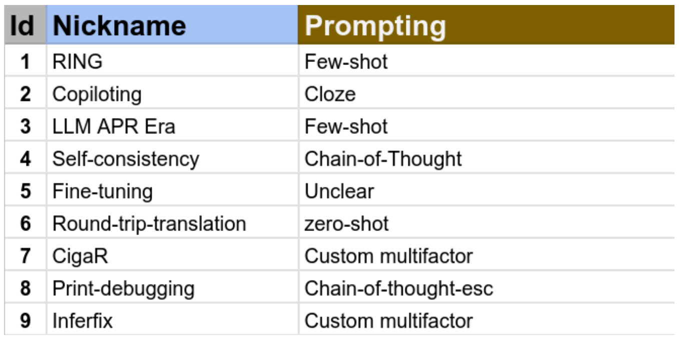
A special case of n-shot learning is so-called *zero-shot learning* where no examples are given. That is, the question is simply asked directly without giving examples. This strategy is applied by Article 6 (Round-trip-translation).

Cloze style prompting is when the task is given to the LLM as an infilling task. In the case of debugging this would be passing a piece of code to the LLM with the faulty lines removed. This way the LLM is tasked with filling in the missing parts. Cloze-style prompting is applied by Article 2 (Copiloting).

Chain-of-thought prompting (CoT) is applied by article 4 (Self-consistency). Article 8 (Print-debugging) also applies a CoT-ish prompting strategy where logs and test cases are passed in the prompt functioning as intermediate steps for the LLM to analyze.

Except for articles 3 (LLM APR ERA) and 5 (Fine-tuning) is the overarching theme for the content of the prompt is that it is very particular, and describing them in detail is outside the scope of this paper. It usually consists of some combination of failed code segment, failed test cases, and console output. Especially articles 7 (CigaR) and 9 (InferFix) use complex multifactor prompting strategies.

Important to note is that it remains unclear from Article 5. (Fine-tuning) of exactly how the LLMs were prompted in their case.



Id	Nickname	Prompting
1	RING	Few-shot
2	Copiloting	Cloze
3	LLM APR Era	Few-shot
4	Self-consistency	Chain-of-Thought
5	Fine-tuning	Unclear
6	Round-trip-translation	zero-shot
7	CigaR	Custom multifactor
8	Print-debugging	Chain-of-thought-esc
9	Inferfix	Custom multifactor

Fig. 6. The prompting strategies applied in the different articles.

D. LLMs used

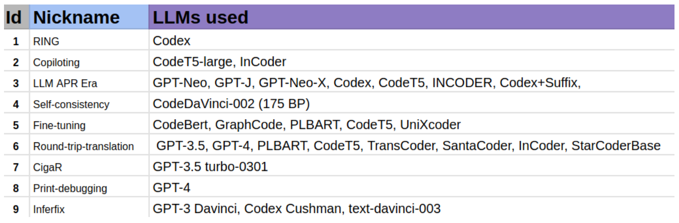
In total a number of 21 unique LLMs have been used across the different articles. In general, there are two approaches when using LLMs for debugging. The first approach is using a general-purpose LLM. The second approach is using a Large Language Model for Code (LLMC). An LLMC is a LLM that has been fine-tuned specifically for programming.

Articles that apply strictly general-purpose LLMs are articles articles 4 (Self-consistency), 7 (CigaR), and 8 (Print-debugging).

Articles that apply strictly LLMCs are articles 1 (RING), 2 (Copiloting), and 5 (Fine-tuning).

Articles that try LLMs in both categories are articles 3 (ERA), 6 (Round-trip-translation), and 9 (Inferfix).

Article 9 (Inferfix) doesn’t only try different models, it uses separate models for different tasks within its debugging process.



Id	Nickname	LLMs used
1	RING	Codex
2	Copiloting	CodeT5-large, InCoder
3	LLM APR Era	GPT-Neo, GPT-J, GPT-Neo-X, Codex, CodeT5, INCODER, Codex+Suffix,
4	Self-consistency	CodeDaVinci-002 (175 BP)
5	Fine-tuning	CodeBert, GraphCode, PLBART, CodeT5, UniXcoder
6	Round-trip-translation	GPT-3.5, GPT-4, PLBART, CodeT5, TransCoder, SantaCoder, InCoder, StarCoderBase
7	CigaR	GPT-3.5 turbo-0301
8	Print-debugging	GPT-4
9	Inferfix	GPT-3 Davinci, Codex Cushman, text-davinci-003

Fig. 7. The LLMs that have been used in the articles.

E. Results of articles

As mentioned earlier the exact numbers are both hard to validate and hard to compare since they are evaluated against different benchmarks under different conditions. Thus we simplify this by paraphrasing the key take-aways. To view the big picture of the results see Fig. 8.

All of the articles claim they have achieved better results than ”state-of-the-art”. Most of the articles compare themselves to other non-LLM program repair tools and how these fair on the same benchmarks that they have tested their LLM-based debugging tool on.

An aspect to take into consideration is the complexity of the bug that is being solved. Articles 5 (Fine-tuning) and 8 (Print-debugging) report on the difference in results depending on the complexity of the bug. In both cases, the repair accuracy drops significantly when LLMs are faced with hard bugs when compared to easy and medium bugs. The other articles don't make this difference which leaves an open question whether the pattern continues in these cases. The difference between *easy*, *medium*, and *hard* bugs is typically defined as whether a fix demands changes to one or several places in the code.

Id	Nickname	Results on easy to medium bugs	Results on hard bugs
1	RING	*Outperforms language specific models*	N/A
2	Copiloting	*Outperforms state-of-the-art + better than base LLM*	N/A
3	LLM APR Era	*Base LLM can substansially outperform existing APR tools*	N/A
4	Self-consistency	*State-of-the-art results*	N/A
5	Fine-tuning	*Finetuned LLMC outperforms state-of-the-art APR tools*	*Repair accuracy drops sharply*
6	Round-trip-translation	*Outperforms state-of-the-art*	N/A
7	CigaR	*State-of-the-art results with reduced cost*	N/A
8	Print-debugging	*Improved results compared to rubber-duck debugging using GPT-4*	*No improvement when faced with hard problems*
9	Inferfix	*Impressive results*	N/A

Fig. 8. Paraphrased versions of how results are described in the articles.

F. Evaluation

The evaluation of the results of the articles have been tested against different benchmarks. See Fig 9. for an overview. Due to LLMs being trained on vast sets of data a key concern is that these benchmarks have been present in their training data thus leading to uncertainty of validity.

The most common benchmark to use between the articles is Defects4J. It is a set of 835 bugs with tests for verifying correct results. Articles 2 (Copiloting), 3 (LLM APR Era), 6 (Round-trip-translation), and 7 (CigaR) all use some subset of Defects4J.

In Article 9 (InferFix) a custom dataset called InferBugs has been created to evaluate the results.

To combat the challenge of data leakage a number of benchmarks have been created with special consideration to the data leakage problem. These, [18], [19], [20], came up in the initial search for this semi-systematic literature review and have not been the focus of this paper, but nonetheless they all try to address the same concern.

Id	Nickname	Evaluation
1	RING	Several benchmarks
2	Copiloting	Defects4J
3	LLM APR Era	Defects4J 1.2 & 2.0, QuixBugs-Python & -Java, ManyBugs
4	Self-consistency	MODIT
5	Fine-tuning	Defects4J
6	Round-trip-translation	Defects4J v12, Defects4J v20, QuixBugs, HumanEval-java
7	CigaR	Defects4J
8	Print-debugging	Leetcode
9	Inferfix	InferredBugs

Fig. 9. The evaluation benchmarks that have been used in the articles.

G. Analysis of Meta-data

Of the nine articles, six make their source code publically available. These are articles 2 (Copiloting), 3 (LLM APR Era), 4 (Self-consistency), 5 (Fine-tuning), 6 (Round-trip-translation), and 7 (CigaR). Articles 1 (RING), 8 (Print-debugging), and 9 (Inferfix) are not publically available.

When it comes to reporting the validity of their findings articles 2 (Copiloting), 3 (LLM APR Era), 5 (Fine-tuning), 6 (Round-trip-translation), and 7 (CigaR) report on validity explicitly. Article 8 (Inferfix) quickly discusses the challenges of evaluating the results due to data leakage. Articles 1 (RING), 4 (Self-consistency), and 9 (Inferfix) have no mention of validity.

When it comes to whether the articles come from academia, industry, or a combination of these there is a good mix of all three. Articles 2 (Copiloting), 3 (LLM APR ERA), 4 (Self-consistency), 5 (Fine-tuning), and 7 (CigaR) all come from research groups from universities around the world. Article 6 (Round-trip-translation) comes from a non-profit research organization/university collaboration. Articles 8 (Print-debugging) and 9 (Inferfix) stem from a joint venture of academia and industry. Article 1 (RING) comes purely from a research team from Microsoft Inc.

Id	Nickname	Public	Validity	Entity
1	RING	0	not reported	Microsoft
2	Copiloting	1	reported	Academia
3	LLM APR Era	1	reported	Academia
4	Self-consistency	1	not reported	Academia
5	Fine-tuning	1	reported	Academia
6	Round-trip-translation	1	reported	Academia/Non-profit
7	CigaR	1	reported	Academia
8	Print-debugging	0	mentioned	Mix
9	Inferfix	0	not reported	Mix

Fig. 10. Meta information about the articles.

V. Discussion

A. A description of the general field for LLM-based debugging

The general situation for LLM-based debugging could be described as being in an early stage due to there being little consensus of what the best approaches are. This is evident from the results of this semi-structured literature review where no study is alike and everyone is experimenting with different ways of achieving better results. Due to the problem where it is hard to tell whether results are due to *memorization* or *generalization*, it is already hard to evaluate how effective LLM-based debugging is. Additionally, there are several other aspects that affect the results, further muddying the field. Due to the current literature trying out so many different variations, it makes it impossible to say with certainty what aspects have the biggest effect on results.

A perspective from which to analyze LLM-based debugging is whether it allows for completely *autonomous* debugging or if it is leaning more towards LLM-*assisted* debugging where humans solve bugs with the help of LLM-based tools. From a design perspective, it can be stated that the current trend is to build more or less completely autonomous systems for debugging. More precisely, the LLM-based debugging tools found within this semi-systematic literature review are set up in such a way that given the right input, they should be able to run without any human interaction. The tool that goes the furthest is Inferfix presented by [11]. Inferfix is integrated into their continuous delivery pipeline and sets out to autonomously fix any broken code for an incoming commit.

However, from a practical standpoint, the current LLM-based debugging tools aren't able to run completely autonomously as none has displayed the ability to solve all bugs without human intervention.

B. *Key design decisions for LLM-based debugging*

There are several key factors that can affect the results of LLM-based debugging. From the semi-structured literature review this paper brings forward 5 key design decisions.

1. The specific large language model (LLM) employed plays a crucial role in its performance [9]. This includes factors such as the size and quality of the training data, as well as whether the training was more general or focused. The number of parameters in the model also influences its capabilities. Generally, a larger model with more parameters trained on a larger dataset tends to outperform smaller models. However, there are exceptions to this rule.

2. Finetuning is an essential technique, particularly for smaller models. It enables them to excel in specific tasks, potentially outperforming larger models when done correctly [10]. This process is especially significant for debugging purposes, as it allows for meticulous adjustments and improvements. Additionally, further finetuning on project-specific data enhances the model's performance and adaptability.

3. Prompt engineering is another critical aspect of utilizing LLMs effectively. Different approaches to formatting information and incorporating external input can significantly impact the model's output and performance. Experimentation with various prompting techniques and information structures can lead to better results in specific contexts. This is clear from the fact that all of the articles that presented concrete tools employed their own specific prompting strategies.

4. Integrating tools to complement the LLM's capabilities can help with tackling the challenges of LLMs. In this literature review, we saw how it was possible to guide the LLM with a completion engine to reduce the number of faulty fixes. Moreover, LLMs can serve as a bridge between different tools, facilitating smoother workflows and more efficient processes.

5. The concept of a thought model, is another fundamental design consideration. Though articles do not refer to this wording in specific, most of them utilize the concept when wanting to improve results. This is especially true for all of the article where the focus is on some LLM-based debugging tool. Emphasis is put on the importance of using more advanced setups beyond simple input-output frameworks. These sophisticated models enable more nuanced understanding and problem-solving, enhancing the effectiveness of debugging tools built upon them.

In the future it will be interesting to see when LLM-based debugging tools will use even more advanced thought models such as Tree of Thought, presented by [8].

C. *Strengths and weaknesses of LLM-based debugging*

When it comes to the strengths and weaknesses of LLM-based debugging it is still too early to make any hard claims. When looking at the big picture of the effectiveness of LLM-based debugging (Fig. 8.) we can see that all studies claim to reach *state-of-the-art or better* results. Though the validity of these claims is hard to validate on a study-per-study basis the fact that everyone reports similar results, suggests that there is some truth to these claims. The fact that the results are similar, while the specifics in each debugging tool are quite different suggests that the underlying technology of LLMs is at the core of these *state-of-the-art or better* results.

As discussed earlier the weaknesses of LLM-based debugging seems to become evident as the complexity of bugs increases. This was not entirely clear from the results of this paper, but what could be deciphered does suggest that LLM-based debugging is hampered when faced with harder bugs. The notion that *harder bugs are harder to solve*, does seem self-evident when looking at it from a logically inferred viewpoint.

Another weakness of LLM-based debugging is the uncertainty and challenge of systematically finding the best way to improve results. More than an inherent weakness of LLM technology it poses a practical challenge for researchers. This is exemplified by a number of points. First, the number of variables affecting the results makes it hard to know the individual effect of each variable. Second, LLM output is nondeterministic, further complicating matters. Third, the generalization vs. memorization problem, means evaluation of results is a challenge.

Lastly, a key strength of LLM can be argued from an economical perspective. Even if it can't solve complex bugs its ability to solve any bug *autonomously* can be hugely beneficial as it frees software developers from wasting time on fixing trivial problems, saving resources for those challenges where they are truly needed.

VI. Validity

The validity of this paper may be considered threatened due to a few factors. First, the complexity of the subject means I have not understood 100 percent of the content in the literature I have reviewed. This is especially true for the underlying transformer architecture used in LLMs. This isn't however the focus of this paper and thus shouldn't affect the findings too much. The second reason that the validity might be threatened is that this paper relies heavily on yet-to-be peer-reviewed articles.

At all stages this fact was underlined and it was explained how this paper mitigates this concern. Due to the focus on current developments, this was however a necessity.

VII. Conclusion

In this paper the current developments of LLM-based debugging has been explained. To achieve this a semi-structured literature review was performed. This involved formulating a search string, gathering articles based on this search string, filtering articles based on certain criteria, and taking standardized notes on all articles. From analyzing these notes the core results of this paper were produced.

From these results, the main findings are three-fold. First, it can be stated that the field of LLM-based debugging is at an early stage with radically different approaches being tested. Research mostly consists of building various LLM-based debugging tools and testing them out. The variety in these tools exemplifies the novel stage of this field.

Second, we have identified 5 key design decisions when creating LLM-based debugging tools. These are 1. The specific LLM used, 2. the Fine-tuning process, 3. prompt engineering, 4. tool integration, and 5. Thought-model.

Third, the strengths and weaknesses of LLM-based debugging can be summarized in three points. 1. LLMs as the core engine behind debugging tools seem to produce *state-of-the-art or better* results across the board. 2. LLM-based debugging tools are effective when faced with simple bugs, but when faced with complex bugs the fix-rate seems to drop rapidly. 3. LLM-based debugging tools can be set up to solve simple bugs almost entirely autonomously creating the potential to save lots of software developer resources.

A key challenge of the research done for this paper was that a lot of the literature used is yet-to-be peer-reviewed. Additionally, any results found in the analyzed articles are inherently hard to validate due to articles often evaluating results on debugging benchmarks that might have been present in LLM training data.

Reading this paper should give a good understanding of how LLM-based debugging is developing right now and what approaches are being tested. It lifts forth the most important design decisions when creating LLM-based debugging tools. It also sheds light on the inherent difficulty of research on this topic. By understanding the current developments, future research can navigate with the big picture in mind, hopefully leading to progress and the avoidance of critical mistakes.

LLM-usage within this paper: The only usage of LLM technology for producing this paper was using ChatGPT-3.5 for Bibtex and LaTex syntax formatting. LLMs have not been used for anything like the generation of text, information extraction, or summarizing content.

References

[1] P. Vaithilingam, T. Zhang, and E. L. Glassman, “Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models,” in *Chi conference on human factors in computing systems extended abstracts*, 2022, pp. 1–7.

[2] A. Zeller, *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017.

[4] H. Li, “Language models: Past, present, and future a language modeling overview, highlighting basic concepts, intuitive explanations, technical achievements, and fundamental challenges.”

[5] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. de Las Casas, L. A. Hendricks, J. Welbl, A. Clark *et al.*, “Training compute-optimal large language models. arxiv,” *arXiv preprint arXiv:2203.15556*, 2022.

[6] A. Karpathy. (2023) [1hr talk] intro to large language models. YouTube.

[7] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, “Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing,” *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[8] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, “Tree of thoughts: Deliberate problem solving with large language models,” 2023.

[9] C. S. Xia, Y. Wei, and L. Zhang, “Automated program repair in the era of large pre-trained language models,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1482–1494.

[10] K. Huang, X. Meng, J. Zhang, Y. Liu, W. Wang, S. Li, and Y. Zhang, “An empirical study on fine-tuning large language models of code for automated program repair,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1162–1174.

[11] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, “Inferfix: End-to-end program repair with llms,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1646–1656.

[12] H. Joshi, J. C. Sanchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radiček, “Repair is nearly generation: Multilingual program repair with llms,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 4, 2023, pp. 5131–5140.

[13] X. Hu, K. Kuang, J. Sun, H. Yang, and F. Wu, “Leveraging print debugging to improve code generation in large language models,” *arXiv preprint arXiv:2401.05319*, 2024.

[14] F. Vallecillos Ruiz, A. Grishina, M. Hort, and L. Moonen, “A novel approach for automatic program repair using round-trip translation with large language models,” *arXiv e-prints*, pp. arXiv–2401, 2024.

[15] Y. Wei, C. S. Xia, and L. Zhang, “Copiloting the copilots: Fusing large language models with completion engines for automated program repair,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 172–184.

[16] T. Ahmed and P. Devanbu, “Better patching using llm prompting, via self-consistency,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1742–1746.

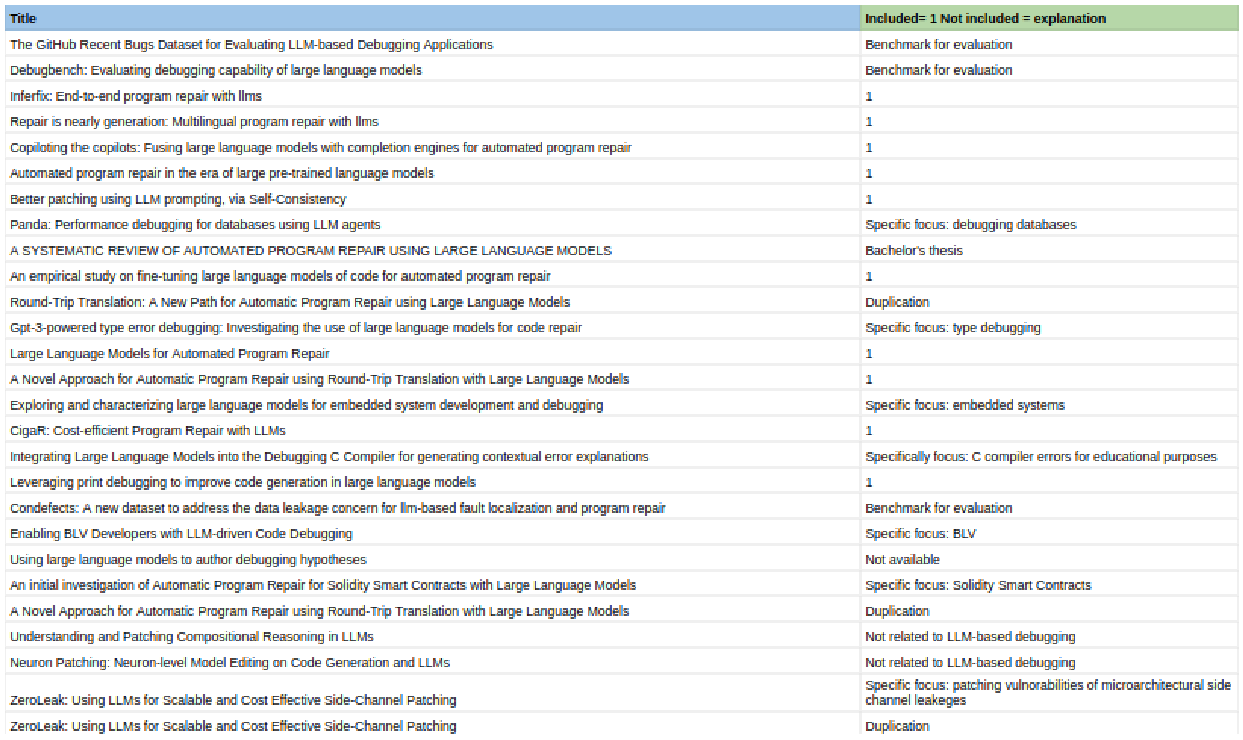
[17] D. Hidvégi, K. Etemadi, S. Bobadilla, and M. Monperrus, “Cigar: Cost-efficient program repair with llms,” *arXiv preprint arXiv:2402.06598*, 2024.

[18] J. Y. Lee, S. Kang, J. Yoon, and S. Yoo, “The github recent bugs dataset for evaluating llm-based debugging applications,” *arXiv preprint arXiv:2310.13229*, 2023.

[19] R. Tian, Y. Ye, Y. Qin, X. Cong, Y. Lin, Z. Liu, and M. Sun, “Debugbench: Evaluating debugging capability of large language models,” *arXiv preprint arXiv:2401.04621*, 2024.

[20] Y. Wu, Z. Li, J. M. Zhang, and Y. Liu, “Condefects: A new dataset to address the data leakage concern for llm-based fault localization and program repair,” *arXiv preprint arXiv:2310.16253*, 2023.

VIII. APENDIX 1



Title	Included= 1 Not included = explanation
The GitHub Recent Bugs Dataset for Evaluating LLM-based Debugging Applications	Benchmark for evaluation
Debugbench: Evaluating debugging capability of large language models	Benchmark for evaluation
Inferfix: End-to-end program repair with llms	1
Repair is nearly generation: Multilingual program repair with llms	1
Copiloting the copilots: Fusing large language models with completion engines for automated program repair	1
Automated program repair in the era of large pre-trained language models	1
Better patching using LLM prompting, via Self-Consistency	1
Panda: Performance debugging for databases using LLM agents	Specific focus: debugging databases
A SYSTEMATIC REVIEW OF AUTOMATED PROGRAM REPAIR USING LARGE LANGUAGE MODELS	Bachelor's thesis
An empirical study on fine-tuning large language models of code for automated program repair	1
Round-Trip Translation: A New Path for Automatic Program Repair using Large Language Models	Duplication
Gpt-3-powered type error debugging: Investigating the use of large language models for code repair	Specific focus: type debugging
Large Language Models for Automated Program Repair	1
A Novel Approach for Automatic Program Repair using Round-Trip Translation with Large Language Models	1
Exploring and characterizing large language models for embedded system development and debugging	Specific focus: embedded systems
CigaR: Cost-efficient Program Repair with LLMs	1
Integrating Large Language Models into the Debugging C Compiler for generating contextual error explanations	Specifically focus: C compiler errors for educational purposes
Leveraging print debugging to improve code generation in large language models	1
Condefects: A new dataset to address the data leakage concern for llm-based fault localization and program repair	Benchmark for evaluation
Enabling BLV Developers with LLM-driven Code Debugging	Specific focus: BLV
Using large language models to author debugging hypotheses	Not available
An initial investigation of Automatic Program Repair for Solidity Smart Contracts with Large Language Models	Specific focus: Solidity Smart Contracts
A Novel Approach for Automatic Program Repair using Round-Trip Translation with Large Language Models	Duplication
Understanding and Patching Compositional Reasoning in LLMs	Not related to LLM-based debugging
Neuron Patching: Neuron-level Model Editing on Code Generation and LLMs	Not related to LLM-based debugging
ZeroLeak: Using LLMs for Scalable and Cost Effective Side-Channel Patching	Specific focus: patching vulnerabilities of microarchitectural side channel leakeges
ZeroLeak: Using LLMs for Scalable and Cost Effective Side-Channel Patching	Duplication

Fig. 11. The title and inclusion/exclusion motivation of all articles found with the search string used for the semi-structured literature review of this paper.