Master's thesis

Master's Programme in Computer Science

Software Testing and LLM-based Systems - a Systematic Literature Review

Vertti Viitanen

February 24, 2025

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Vertti Viitanen

Työn nimi — Arbetets titel — Title

Software Testing and LLM-based Systems - a Systematic Literature Review

Ohjaajat — Handledare — Supervisors

Prof. Jukka Nurminen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	February 24, 2025	32 pages, 2 appendix pages

Tiivistelmä — Referat — Abstract

More and more software is integrating Large Language Models (LLM) into their functionality. Some of the tasks the LLMs perform are in areas where incorrect behavior can cause a lot of problems for their creator. This is why this thesis conducts a systematic literature review on software testing LLM-based systems and software.

The initial criteria included only two papers, which is why the scope was slightly expanded to include testing LLMs in general without changing the search string that was used to retrieve the studies from the databases. Due to this change, the final number of studies reviewed became 13. The contributions varied from experience reports, to models to actual tools.

Most of the issues, solutions and often terminlogies had no overlap between the studies. The two most common that did overlap were the problem of nondeterminism and the importance of prompting.

From collating all of the issues and proposed solutions, we suggested some fruitful avenues of future research. Prompting tooling, repeating this review with search thing that considers the expanded scope, moving towards common vocalbularity and seeing how the demands of testing LLMs change in respect to the complexity of the task were the topics we highlighted the most. However, due to the scarcity of work on the issue of testing LLM-based software, any research will most likely be of value.

ACM Computing Classification System (CCS)
Software and its engineering → Software creation and management
→ Software verification and validation

Avainsanat — Nyckelord — Keywords

software testing, large language models, systematic literature review

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software study track

Contents

1 Introduction

Noting the rise in interest in Large Language Models (LLM) is something of a cliche at this point. Multiple studies bring this up in their introduction, both from the academic and public perspective (Laakso, 2023) (Jabborov et al., 2023). However, this is not without reason. Generative AI, many of which have LLMs as their backbone, has fully broken into the public consciousness. We would be willing to wager that few people in the modern world have not tried interacting with them in some fashion. In addition, this kind of software is starting to get integrated into various industries. And again, this is not without reason. The market size for Artifical Intelligence is expected to reach over 1300 billion dollars within the next half decade (*22 Top AI Statistics And Trends In 2025* 2024).

'Move fast and break things' is a motto coined by Mark Zuckerberg. It is a mentality which urges developers to accept possible risks with new frontiers in the hopes of being an early adopter. While the man himself no longer believes in it, similar mindset can still be seen in the technology sector today, and AI systems are not immune. Chatbot and other implementations have seen massive adoption rates (*22 Top AI Statistics And Trends In 2025* 2024). Things have also been broken. While the outcome in it is not as drastic as some other cases, the example we feel best illustrates this is Air Canada having to honor a refund policy hallucinated by their chatbot (*Oh, Air Canada! Airline pays out after AI accident* 2024). This we feel is completely reasonable: if a bussiness hired animals who misbehaved, the bussiness would be responsible. The logic seems to be the same with AI.

But the question arises, how can bussiness avoid this happening. Software Testing is the field that currently tries to diminish the issues that could arise with software malfunctioning. However, the methods of testing traditional software are very different than those of intelligent software. But can we go deeper still? LLMs are the technology underlying many of these new implementations. Is there any unique challenges or possibilities, when testing systems that are based on LLMs. We have conducted a systematic literature review to find an answer for these questions.

This review is split into a few larger sections. Following this introduction, we will provide background information into the technical aspects of LLMs, software testing and the difference between LLM testing and system testing. Then we will go over the methodology of the review itself, so the reasoning and steps we took are clear and can be replicated if

need be. The section that follows will discuss the studies reviewed first as a whole and then from the perspective of the research questions. Before the final conclusions are reached, we will discuss potential steps to take in the future and any vulnerabilities in the study we have conducted.

1.1 The use of LLMs this study

While originally we felt it would be somewhat hypocritical to use the aid of LLMs in a study, that is in a way dedicated to finding their defects. However, since many of the tools that we have used in our writing have integrated some LLM features, most notably the grammar correction in Overleaf, which uses Writefull's LLM, we decided to allow more leeway in our use. In addition to this use, we utilized Google's Gemini chatbot to search material that we could cite us sources in the Background chapter. No information provided directly by a AI was used in this thesis.

2 Background

We consider the material presented in the Results chapter to be understandable without a deep technical understanding of software testing or large language models. This chapter will offer context for some of the more basic concepts of both fields.

2.1 Large Language Models

2.1.1 Structure and underlying logic

Language models are a tool used in natural language processing (NLP). They are statistical representations of a language that tell the probability that a given sentence exists in the modeled language (Luitse and Denkena, 2021). Neither language or sentence in this context necessarily means the same as it would with naturally spoken languages: the language could be a programming language and the sentence can be just a word or symbol. More generally speaking, the probability represents the likelyhood that a sequence could originate from the training material. The useful part of this is that the model can predict what the next word in a sentence would be, and thus it can generate text (Luitse and Denkena, 2021). This prediction also depends on the context window the model is using. For example, for the phrase "deus ex" and a context window of two words, the answer would most likely be "machina", but for one-word window, where only the word 'ex' is within context, responses like "boyfriend" or "girlfriend" would be more likely (Laakso, 2023).

Although one of the studies we reviewed (S12) sets the threshold for LLMs at a billion neural network parameters, there is no specific size where a language model becomes "large". Despite this, the term is still useful. The aforementioned context window, the training data, and the number of parameters in the neural network technology underpinning modern language models: with all of these variables a larger size has been shown to produce better results Luitse and Denkena, 2021. This property is called *emergence*: just as the water in a drinking glass is made from the same molecules as the ocean, the latter exhibits much more complex phenomena sheerly due to its size. The transformer technology, which is in the name GPT (Generative Pre-trained Transformer) that can be seen in multiple AI

applications, has facilitated this growth even further. Unlike previous architectures such as recurrent neural networks and convoluted neural networks, transformer architecture allows training of the model in much shorter time even on larger datasets due to increased parallelization (Luitse and Denkena, 2021). This is possible because instead of using recurrence to get the model to predict the next word in a sentence with a context window of greater than one word, transformers recognize connections between all the words in the input and identifies the most relevant ones.

The "generative pre-training" part of the GPT acronym refers to the methodology of how the model uses the training data. It is a semi-supervised learning method, where the model first learns from a large corpus of unlabeled text until it can generate that type of text(Radford and Narasimhan, 2018). Only after that the model is adapted for discriminative tasks with labeled data. Since labeling the data is slow and expensive, this approach allows for scaling the training data much more efficiently.

2.1.2 Prompting

Prompts are natural language inputs that are the interface via which developers interact with LLMs. Instead of building specific ML models for each task, the prompt and the LLM together can provide equivalent answers, which significantly increases prototyping speed (Liu et al., 2021). Some of those models might not even be possible to build, since the labeled data for their creating simply does not exist in large enough amounts. The methodology for choosing the prompts that perform the desired task most effectively is called *prompt engineering*.

2.2 Software Testing

Software testing is a process that verifies and validates that the program works as intended, and detects any defects and identified the ones that are serious and need to be fixed (Bentley, 2005). Verification and validation are two distinct processes that often get confused. Verification ensures that software meets its technical specifications: outputs are formatted as expected under specified inputs and other conditions. For example, a query must return the correct number of fields and they must be sorted by yet another field. Validation concerns itself with bussiness requirements. In other words, validating ensures that the software does what it is actually needed to do, even if it otherwise "works". Defects

are any discrepancy between expectations and outcomes; scenarios where the software does not behave correctly. Software testing also categorizes defects by their severity. For one to be serious, it must affect usability or functionality from the customer's perspective (Bentley, 2005). For machine learning software however, the models are expected to make occasional mistakes, especially in classification tasks Kästner, 2025. That is why instead of correctness, aggregated modes of evaluation like accuracy are used instead.

For software testing to be effective, it must be done continuously throughout the development lifecycle (Bentley, 2005). The testing has different levels which relate to differing development phases. Unit testing is the lowest level of testing and examines the individual, smallest level modules of the software. It focuses on reliability and functionality, and it is done in a testing environment before the component is integrated into the system. Quite logically, the next level of testing is system testing, where all the modules that comprise the application are tested together. The goal is to detect defects at the border of the individual modules. This requires many test runs since a lot of different features need to be validated using both normal and incorrect inputs. System testing also avoid outside influeence, i.e. other systems. The presence of any other necessary systems are tested during the next level, integration testing. The highest level of testing is user acceptance testing, where the application moves to the hands of the users. At this point the developers have resolved all the identified defects, and while in an ideal world the users would not find any more, this rarely the case in practice.

Whenever a defect is resolved, a test verifying the fix should be created. These tests ensure that any changes in the future do not reintroduce the defect back into the application. These types of tests are appropriately called regression tests.

3 Methods

This chapter will focus on how we conducted the search itself and the reasoning behind the choices we made during the process. This includes giving an overview of the review conducted, the research questions and their evolution during the process, why the databases we used were chosen, the formation of the search string, and the criteria for inclusion and exclusion. We will also go over the changes made to each of these that were caused by the small number of studies the initial choices gathered. Finally, this chapter will cover the number of studies handled during the process and how each step changed that number.

3.1 Review

We conducted the systematic literature review by using the process steps outlined by Mäntylä in *Mini Systematic Review Guide for Masters's thesis* 2023. We mostly focused on the steps that outlined the actual process, since the guide was created with a smaller scope in mind. In addition to the steps provided in the guide, we considered doing a quality evaluation on the selected papers. However, this did not end up being necessary as the initial number of papers ended up being only two. During the discussion portion, we discuss the quality of some of the individual papers.

3.2 Research Questions

The general question that lead to this study was 'What does software testing mean, when the software in some way relies on large language models?'. Software testing often measures correctness, and how can such a thing be defined for something with as varied outputs as an LLM. These musings lead us to the following two research questions:

- *RQ1*: What does existing literature say about the challenges of testing LLM-based software compared to traditional software?
- *RQ2*: What methods and results does existing literature have for testing LLM-based software?

However, since the initial result of two studies was not broad enough to provide a trustworthy answer to these questions, the question needed to be broadened. While we still consider the initial research questions, we added a third research question to address the spirit of our general question.

- *RQ3*: What notable aspects from the perspective of software texting do the reviewed studies bring up?

An important distinction we want to make, is that this question does not ask how LLMs could be used in Software Testing. Our interest is in how the presence of LLMs affects the testing, not how all testing could change thanks to the models.

3.3 Databases

We used three databases to conduct the search: Scopus, the Association for Computing Machinery Digital Library (ACM DL), and the Institute of Electrical and Electronics Engineers Xplore (IEEE Xplore). All of these databases we consider to be well respected and readily accessible to university students. Most of them also specialize in computer science. Google Scholar was also considered, but we dropped it after initially testing the search string. We considered three databases to be enough for a single person review, and we preferred more primary sources, where Google Scholar is a meta-search tool which spans multiple places of scholarly literature (*About Google Scholar* 2025). Among the sources Google Scholar uses to pull studies from are the three databases we are already using, further reducing the need to use Google Scholar as a part of the study.

When we found the same study in multiple databases, we prioritized ACM DL over IEEE Xplore, and Scopus over ACM DL. Our hypothesis was the broader the topics the database covered, the more likely it was that a later version of the study would be published there, if any modifications had been made.

3.4 Search String

The search string we chose was heavily inspired by the one used in 'Taxonomy of Quality Assessment for Intelligent Software Systems: A Systematic Literature Review' (Jabborov et al., 2023). They achieved their query by searching databases with keywords synthesized

```
('LLM software' OR
'large language model software' OR
'LLM-based software' OR
'LLM-based system' OR
'LLM-powered software' OR
'LLM-powered system')
AND
('software testing')
```

Figure 3.1: The search string

from their research questions and tallied the number of hits from each site. Using these results, they eventually settled on a search string. Our string was achieved by utlizing the structure of their string and replacing their topic nouns with those of our own. The result can be seen in Figure 3.1.

We observed from the aforementioned taxonomy study that when the topic was part of a hyphenated compound word, only the abbreviated version was necessary. That is why 'LLM software' and 'large language model software' are the only query pair with both versions of the word. The one part of our search string that was not achieved via this replacement method was the inclusion of the pair of phraes with the '-powered' suffix. This is because this paper originally included that phrase as part of the title, and a cursory search from the selected databases provided some results. We did not use any wild card characters in the search string because the words we used are not conjugated that often apart from pluralization, and we deemed that the databases' search tools could handle that even without the wild card character.

The search string was used on each of the selected databases with some semantic chases to satisfy each site's differing syntax. The search was performed against the full text and all meta-data of the studies, which includes the abstract, the author-chosen keywords, and the sites' indexing terms.

The final results for Scopus and ACM were obtained via this process. However, we deemed the initial number of results obtained from the IEEE, 1748, to be too high. Many of the hits also seemed incidental. For example, only matching because of the name of the conference.

First, we removed the final part of the query '**AND** (software testing)' and instead utilized IEEE's publication topics filter. This meant that the search string used the same parts for LLMs as for the other two databases, but restricting the studies to only consider topics related to software testing was done by IEEE's internal sorting. This change reduced the number of results to a tenth, from 1748 to 161. At this point, we noticed that even if we included the phrases within the quotations, there were still partial results in the matched studies. For example, even though we searched for Large Language Models, papers concerning Unified Modeling Language were found. Due to this observation, we conducted the next search on IEEE only on the abstract of the papers, instead of the full text and meta-data. However, the full 161 articles were included for consideration for the second iteration of the research questions and inclusion and exclusion criteria.

3.5 Inclusion and exclusion criteria

For a study to be included in the final review, it must meet each inclusion criterion and not meet any of the exclusion criteria.

- *IC1*: The study is in English.
- *IC2*: The study is peer-reviewed.
- *IC3*: The study is a conference paper, article, or a review.
- *IC4*: The study is about LLM-based software.
- *IC5*: The focus is on LLMs.
- *IC6*: The study tests LLMs in some capacity.
- *EC1*: The study is behind paywall, or otherwise inaccessible via reasonable means for a university student.
- *EC2*: The study focuses on how software testing could use LLMs.

The reason for each criterion can be neatly divided into two categories: Limiting the type of literature we have to review and answering the research questions. IC1-3 and EC1 are of the former category, which explains why they are so topic-independent. Since this study is written in english, to keep the replication possibility open to as many readers as possible,

we decided to exclude non-english studies. Our reasoning was similar for the creation of EC1. As we are not conducting a review on gray material, the constraints on the type of study were included and the requirement of being peer-reviewed. We could not always be certain of the latter, so we used the publication as a heuristic. If the source the study was featured in seemed respectable, we decided it was safe to assume that the paper would be peer-reviewed.

The remaining criteria are of the second category, answering the research questions. IC4 and IC5 refine the studies to answer RQ1 and RQ2. The remaining criteria were created after the initial application of the inclusion and exclusion criteria on the studies found. We created them to answer RQ3 and thus replaced IC4 and IC5 for the second filtering, as they were broad enough to encompass the studies that those two criteria would have allowed.

3.6 Search Outcomes

We conducted the search on the databases on November 6th 2024. We gathered the results into a spreadsheet with a link to the publication. Then we started applying the inclusion and exclusion criteria on the papers. After we noticed that the criteria only allowed two papers through, we added the additional research question and criteria, as previously discussed. On December 5th 2024, we had arrived at the final resulted papers, at which point we downloaded a copy of each of them to ensure no further changes could end up in them. The way each step of the process affected the number of papers is depicted in Figure 3.2. All the steps in the search process produced a final result of 13 studies, the details of which we will discuss in the results section. The figure does not depict EC1, as the selection of databases meant that no papers were filtered because of it.

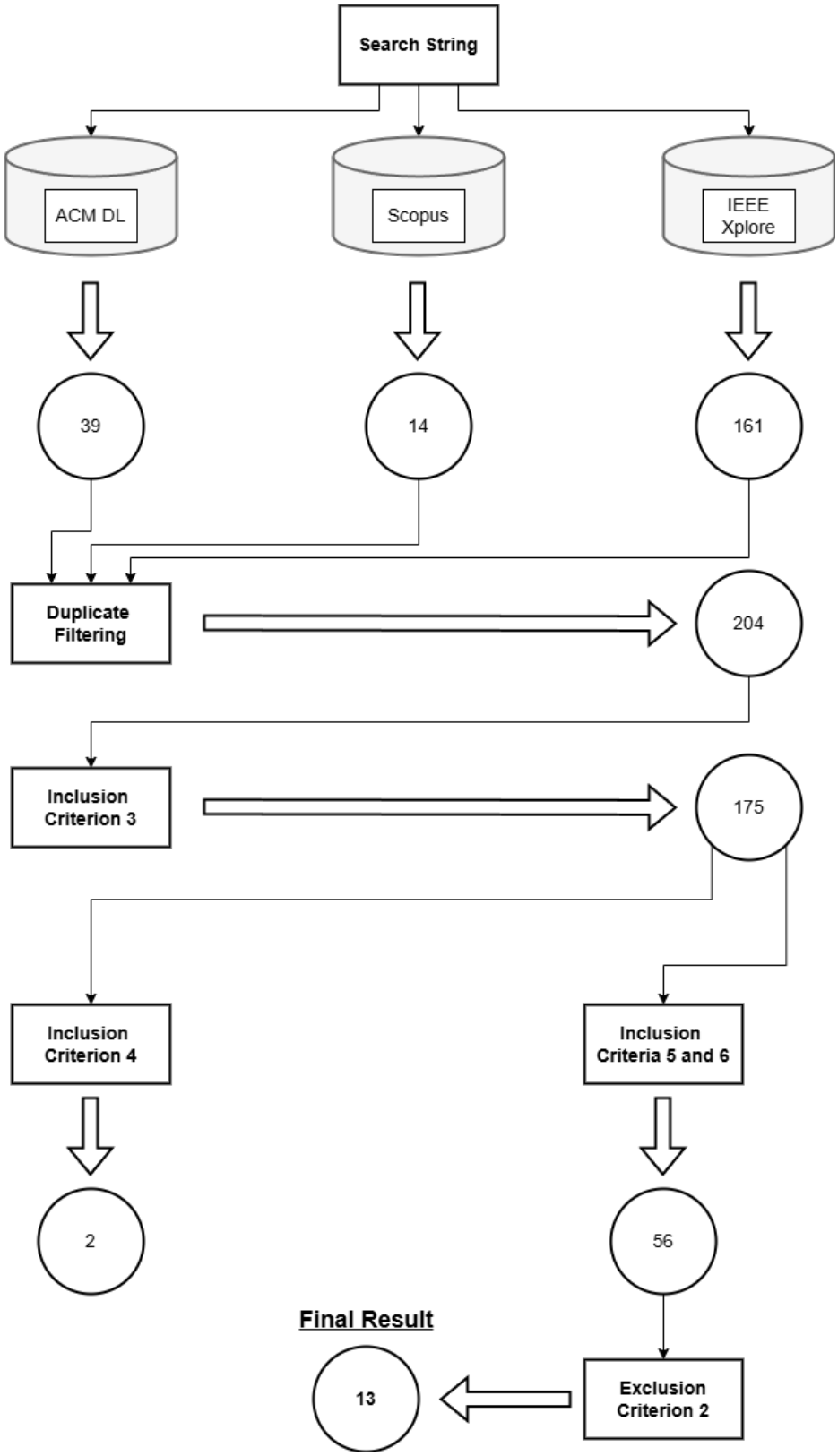

Figure 3.2: Amount of studies during the search process

4 Results

This chapter will give an overview of the selected studies and the relevant data we extracted from them. In addition, we will answer the research questions defined in Chapter 3.

From this point forward, we refer to each of the reviewed studies with an index code. This code consists of the prefix S and a zero-padded running number from 1 to 13, creating codes from S01 to S13. We assigned the number to each study according to the alphabetical order of the title. We used this method in lieu of sorting the studies by release year, which is the common way to encode studies in a systematic literature review. We did this due to the close release dates of the chosen studies.

4.1 Overview of the studies

The first matter we will discuss is the results of the original search, due the cascading effects it has. Before we amended the inclusion and exclusion criteria, the process left us with only two papers. Assuming our results themselves are valid, we propose that low yield either means that there is not yet a standardized way to refer to software that somehow utilize the LLMs, or that using intelligent systems as a part of software is such a recent area of study that there simply is not enough scientific literature to conduct a review on the topic, especially when the focus is narrowed to LLMs. However, even after we expanded the topic to cover LLM testing in general, the number of studies included increased to only 13. We think a partial reason for the low yield is that we still used the original search string. Still, we chose not to change the search string to keep the focus of the review on LLM-based systems, or at the very least to be able to highlight the potential lack of research on the topic.

A noteworthy matter that points toward greenfield explanation is how recent all of the reviewed studies were. Except for S12, every single one of the 13 studies that made it through the inclusion and exclusion step was released in 2024. Even S12 was outside this window by little more than a week, since it was released in late December 2023. This spread of dates also means that each study was released within a year of the date we used

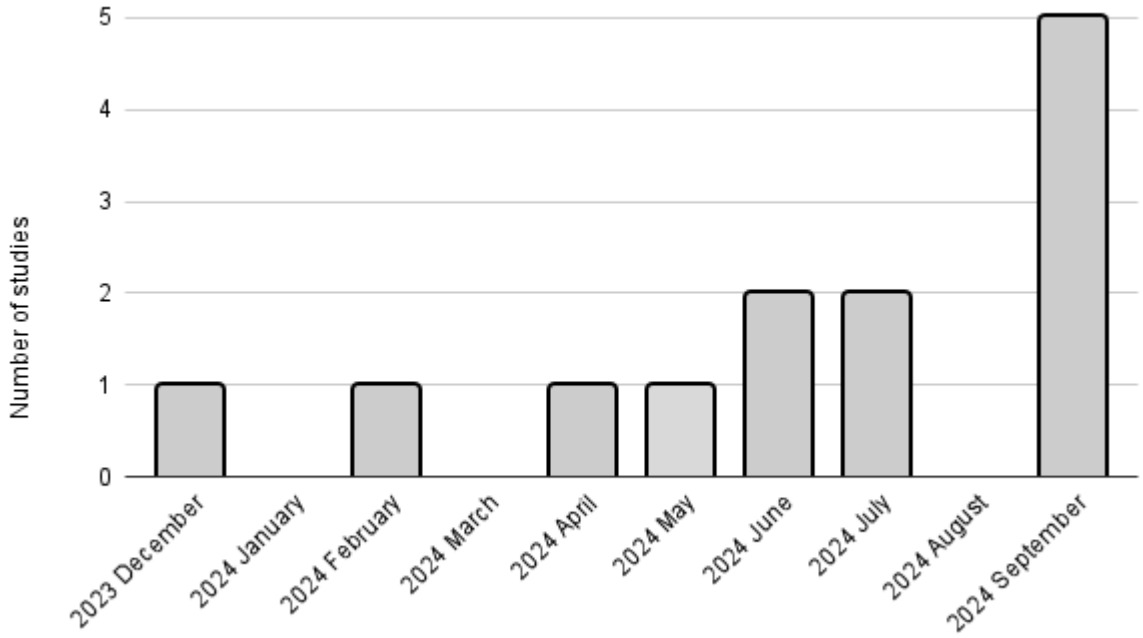

Figure 4.1: The number of studies per month

the search string on the databases. The way in which the release dates spread throughout this period is illustrated in Figure 4.1. The trend in the number of studies increases towards the end of the year. The low number of studies reviewed is not enough to make any statement in itself. Still, this trend, combined with the fact that a low number was all that could be gathered, together do point to how new the topic under research is. The number of releases towards the end of the year seems to start to accelerate after the 17th IEEE International Conference on Software Testing Verification and Validation that was held late May 2024 (*ICST 2024* 2024). However, only two of the reviewed studies were published there.

Another observation we want to raise is the number of studies that deal directly with code. Of the 13 studies, 4 dealt with LLMs generating code (S03, S04, S05, S09) and 2 with the ability of LLMs to read code and discern something from it (S07, S08). For the other studies, no topic was shared and they varied wildly, from what ethical biases LLMs have (S01) to assessing how reliably they can tell that a given driving scenario is realistic (S10). However, this result does not surprise us. As computer science is the field from which LLMs emerged, it is also a natural target for the promises of its creation. The fact that Figure 3.2 shows that the number of studies excluded by EC2 is more than three times greater than the number of studies reviewed also supports this hypothesis.

4.2 LLM integrated software specific issues

To discuss the challenges of testing LLM-based software instead of only LLMs, we will focus on studies S11 and S13. These are the two studies that made it into the review process when the inclusion and exlusion criteria were formed from only RQ1 and RQ2. They both deal with some LLM-based software, and while S11 is not explicitly about software testing, it does raise challenges relevant to the topic.

4.2.1 Retrieval Augmented Generation systems

S11 presents three case studies regarding Retrieval Augmented Generation (RAG) systems. From these findings, they categorize seven different failure points of these types of systems.

RAG systems offer an option to an area where LLMs struggle, up to date information and domain specific knowledge that you might want to use but is not publicly accessible. Instead of training a new LLM for these purposes, in RAG systems you integrate retrieval mechanisms around an existing LLM, and limit its instruction to only generate answers based on the context the retrieval gives it. For example, as a company you might create a RAG system for some large internal document. In theory, it should be able to give you answers exclusively based on that documentation, and answer 'Sorry, I do not know' in the happy path where the information simply does not exist inside the documentation. This eliminates the need for retraining LLMs when the information you want it to base its generation on changes.

Although S11 mentions software testing only as an area of further study, some of the points raised are relevant. One of the key takeaways from the study is that '*the validation of a RAG system is only feasible during operation*'. This is due to the expense, latency, and the varying performance between LLM releases. Literature regarding these problems exist for general machine learning, but such adjustments have not been applied to any LLM-based systems.

When testing RAGs specifically, the seven identified failure points could be used as a guide when creating the tests. Regarding these tests, a RAG specific issue is the generation of the questions and answers that are needed to test the system. S11 notes that, due to other work on the topic, generating these could also be done with LLMs. This is because the questions and answers needed, based on often unindexed documents, might not otherwise be available. The issue remains open.

4.2.2 LLMs through an API

S13, the most relevant study found for the topic of testing LLM-based software, is about regression testing software that utilizes LLMs through an API.

The study recognizes that the high cost of LLMs means that many applications will instead rely on LLMs provided by other companies. However, that means that the creators of those applications might not be able to choose which version of the LLM to use. Even worse, some of these updates are done silently, without the change being noticable downstream via the API signature. Although it is not an issue exclusive to LLM-based systems, S01 raised silent versioning as an issue.

The issue is not merely theoretical, as S13 provides a case study where different versions of an LLM cause performance downgrades. Even if LLM providers would not update their models silently, the same issues with version changes are still present. This is because specific LLMs will inevitably become discontinued and deprecated.

S13 also notes that these APIs evolve over time and this change is often not documented. Other literature points this happening in some of the more common LLM providers like ChatGPT and Amazon Rekognition. However, this type of undocumented changes is also found with traditional software API and other, non-LLM ML API.

4.3 LLM testing issues

The purpose of this review was to identify challenges in LLM-based software. However, the literature on the topic is very scarce. Therefore, we will also discuss some elements of testing LLMs directly. We felt this was appropriate, since even though S13 deals with LLM-based software, most of the testing challenges and solutions offered are applicable to LLMs in general. Conversely, we propose that this means that using other studies to gather solutions and issues regarding testing LLMs is likely useful when considering testing LLM-based software. Still, the type of solutions we highlight in this section are not likely to provide solutions to the issues raised in Section 4.2, which are specific to LLM-based systems.

4.3.1 Nondeterminism

The most common issue raised in all reviewed studies is nondeterminism. Seven out of the thirteen studies adjusted their methodology or identified the problems caused by LLMs not always providing the same output for the same input (S01, S03, S05, S06, S08, S10, S13). This is done intentionally by the LLM developers to provide higher quality results (S13), and to foster creativity and diversity to the answers. This issue also extends to research on LLMs, as replicating results now involves an element of chance (S03).

To allow users to tailor the variance to their needs, most LLMs include a hyperparameter (S03) called *temperature* which determines how random the generated answers should be. Even if the parameter exists in an LLM, it does not necessarily follow that the user has access to it, and most LLM queries default to higher temperatures (S06). If the customization of temperature is available, in theory it should mean that LLM could provide deterministic results. It is a common belief that this is the case for temperature zero, the lowest setting (S03). This belief exist at least for common LLMs such as ChatGPT(S04), and one of the reviewed studies also stated it to be true (S01). Despite this, multiple of the reviewed studies claimed that LLMs are nondeterministic even with a temperature of zero (S03, S10, S13). In fact, non-determinism of ChatGPT was the central topic in S03. They found that the nondeterminism at temperature of 0 scales with the input length of the query. Still, temperature is not completely irrelevant as lower temperatures do provide more predictable answers (S08) even if they are not truly nonrandom.

In S13 the researches noted that which temperature is used varies based on the task LLM works on. The study also stated that LLMs that perform generative tasks instead of classification tasks tend to often use higher temperature.

4.3.2 Prompting

Prompts are the natural language input is provided to an LLM. Apart from S02, which focused more on software testing in general, all of the studies reviewed mentioned prompt design as some part of their process.

Prompting interfaces allows for more rapid prototyping and iteration with LLMs. S13 illustrates this by stating that an LLM and a prompt together can be thought of as comparable system to traditional ML model that was trained on a specific task. Developers are able to tweak and validate prompts with only a few examples, avoiding the need to

for data curation and similar tasks that the traditional ML models would have required However, prompts introduce a kind of brittleness to the development process. A small change in prompt can alter the output greatly. This also means that the same prompt can produce wildly different results on a different LLM, or even a different version of the same LLM. This brittleness is the reason why changing versions are such a threat to LLM-based software.

A whole field of prompt engineering has emerged due to the relationship between LLMs and prompts. Testing LLMs is also multi-disciplinary field to begin with, and the fact that prompting itself can be a specialty complicates the process further (S01). Different prompting techniques have emerged, such as the difference between zero-shot and few-shot prompting, which specify how many examples or demonstration you provide the LLM within the prompt. Where the example is within the prompt can also affect the results (S13).

Two of the reviewed studies found the current tools and systems for prompt engineering to be lacking (S06, S13). On the users side, the focus on narrow domains and the tools being closed source were identified as an issue. Additionally most of the existing tools require programming knowledge, exacerbeting the problem of multi-disciplinarity as well. From the testing side of things, the existing systems do not support the need for monitoring and versioning prompts that testing LLMs require. Since the creation process of the prompts themselves is an iterative one, prompt versioning or a *prompt history* could make sure that the lessons learned would not be lost.

4.3.3 Closed LLMs and Black Boxes

Considerable number of LLMs are not open source. S12 states that the main reason for this are the computational and storage requirement, and we feel the need to point out competitive advantage. This can cause a major issue. S12 states that wider research and industrial applications are more likely to use open LLMs, which means the standardized intelligent systems are more likely to arise from this sector. However, initial research is more likely to use the wider known closed source LLMs. Generalizing the results between these two types of LLM is not proper (S12). Major reason for this is scale. LLMs are already a product of emergence, and the difference in the amount of parameters between LLMs and what S12 calls Super LLMs can be as large as the gulf between LLMs and traditional language models. GPT-3 has 175 billion parameters (Brown et al., 2020), and

GPT-4 is rumored to have 10 times more*, where as open LLMs do not break the 20 billion limit (S12). Traditional LMs linger below the 1 billion mark. S12 compared traditional LMs to canoes, LLMs to yachts and Super LLMs to cargo ships. We feel this illustrates the differences in scales quite clearly.

LLMs are built on deep neural networks and as such are considered to be black boxes (S12). In addition, information about their specification is not readily available. This means that the testing of LLMs falls almost directly into the realm of functional testing, which is quite limited compared to the options available to traditional software. S13 also points towards this reality: it is the only study reviewed with explicit type of testing done on an LLM, and the type chosen was regression testing, which is a subgroup of functional testing.

4.3.4 Correctness

Correctness is defined as error free operation (S02), which is a rather binary concept. For both classification and generation tasks, LLMs lack easily definable and objective corrects standards for correctness. Additionally, performance graduation with LLMs is gradual rather than sharp, so the binary approach captures less of the systems quality.

If correctness is difficult to define, how can it be said that a system has experienced quality loss. Individual predictions can regress often (S13), so treating each of them as a marker very quickly becomes unmanageable. On the other hand, looking at aggregate data can be too general and may lose the important details which would allow to developers to attempt to correct the issues.

4.4 Proposed solutions

Many of the reviewed studies offered potential methods for enchancing the results provided by LLMs. Others used novel methods or applied traditional methods to directly test some capability of LLMs. This section focuses on this aspect of the papers. If a situation arises where one study brings up issues in the suggestions provided in another, that too will be mentioned in the following sections.

*OpenAPI has not released any open studies about GPT-4, and the sources we could find about the number of parameters were ones we did not feel comfortable using as citations.

4.4.1 Better tooling

As some of the problems, mainly dealing with prompts, are caused by insufficient tools, the trivial solution is to create better ones. Still, S06 was the only reviewed study for which the contribution was a tool.

The tool released by the study is called ChainForge, an open source visual interface for prompt engineering. It aims to be as accessible and usable as chat interface while providing the opportunity for evaluation and testing, without limiting the possiblity of exploration which is what other graphical tools focus on more. Providing more tools for prompt engineering that require less technical background can help in the multi-disciplinarity issue of testing LLms. From our perspective however, the most important feature of ChainForge is the ability to simultaneously send multiple queries with slight parameter changes, and possibly send those queries to different LLMs at the same time. This kind of feature can definitely be useful when testing how the performance of prompts change between versions, which paramount for detecting regression from LLM updates. However, it does not seem that ChainForge offers much in the way of versioning these various prompt, thus not aiding the field with the issue of *prompt history*.

4.4.2 Software Interrogation and Diagnosis

S02 was the reviewed study that strayed furthest from LLMs, but still provides useful insights. They argue that testing processes should be thought of as 'data generators for the active learning of software system models'. This would reframe testing as an iterative process that would generate data, which can be used to create models that enable high value analysis. Practically, this would mean extending from merely ensuring correctness to software interrogation and diagnosis. The downside of this is that software testing itself would become more cross disciplinary, since the testing process would start to include the work of statisticians and domain scientists. The authors of S02 consider this a reasonable tradeoff and demonstrate the positive aspects of the approach with two case studies.

Software interrogation asks questions about how the software is running, and through the answers tries to improve its performance. Even though the term seems to be novel to S02 and contains some complex ideas, the big difference between it and monitoring and testing in general is that its an active process that is done to even correctly behaving, defect free software. The goal is to identify problematic outputs in situations where measures of correctness might not exist. In the case study, for the problem of linking smaller DNA

snippets to larger genome in the database. By comparing the results of different classifiers, especially the parts where the results do not match, they were able to identify cliques in the data. These could be used to prematurely terminate searches in the reference database. Even with a conservative strategy, they managed to reduce computational burden by more than half. However, they do note that these types of strategies are both conditional and not risk-free. Still, these kind of exclusions become less risky the more the user has experience with the software.

Software diagnosis deals with the results of a software, and identifying the cause of potential problems. The second case study compared different classifiers, and identified a boundary for their answers. By comparing the boundaries, they deduced that while the algorithms vary in performance, the varying fragility in reads was caused simply by the difficulty of classification. More generally, they could learn about robustness of differing systems, but specifically for the input data. From an LLM perspective, the interesting aspect of this type of approach is the methods dealing with accuracy, trustworthiness, and surrogate uncertainty measures. These are aspects which could be used to approach LLMs, that report to their user when their answers are near their boundary of trustworthiness.

4.4.3 LangBiTe

LangBiTe is an approach to modeling ethical requirements and evaluating their fulfillment by LLMs (S01). Although the model is for a specific domain, and the name itself is short for Large **Lang**uage Model **Bi**as **Te**sting, we see no reason why this model could not be extended to other, non-ethical concerns.

LangBite is split into four stages: requirement specification, test generation, test execution, and reporting.

In the requirement specification stage, the requirement is modeled with a domain-specific language. For ethical requirements, this identifies the ethical concern and affected communities, but the generally applicable parameters are tolerance and delta. The tolerance value sets the rate of passing tests that the LLM needs to achieve in order to meet the requirement. Delta specifies the variance allowed when asking numerical questions about differing subgroups. S01 notes that testers should be careful when the variance reflects actual inequality, instead of just bias in the LLM, since in these scenarios the values in fact should differ. However, the study does not elaborate on what one should do in these cases.

In test generation, the designer defines test scenarios that will be represented by different prompt templates. Prompt templates are pieces of text, where communities (which we think can be extended to arbitrary groupings) linked to the ethical concern are inserted into the template. For example, 'Is **A** better than **B**' would replace the bracketed portions with linked communities. Important to note is that this prompt generates two cases, one where **A** is replaced with value **X** and **B** with value **Y**, and vice versa. In addition, in the generation step the temperature of the LLM, what LLM is used, and the number of test cases generated for each requirement are decided. The following test execution step simply refers to sending the created prompt to the LLM and storing the response.

In the reporting stage, the answers from each same template are grouped together and compared to the answers of a test oracle. Each template has its own test oracle and the complexity of an oracle can vary wildly. For simple yes-or-no questions, it can be only a single word. On the other hand, or some questions, building an oracle of any kind is not feasible. In these situations, S01 suggests using another LLM as a judge. For each template that maps the same concern, the number of answers that match the answer of the oracle for the template is tallied. If the ratio of these answers to total answers exceeds the tolerance value set for the concern, the LLM is considered to not exhibit the bias the requirement models.

The model is very extensible, easily supporting more concerns, adding more groups to existing concerns and changing the oracles. Even though S01 views LangBiTe as exclusively for detecting biases, we feel that nothing in the process locks it specifically to those tasks, and have aimed to demonstrate that in the description.

4.4.4 Assured LLM-based software engineering

Many of the reviewed studies approached LLMs from the angle of code generation. But when it is done without human supervision, how can we be sure that the properties of the original code are not regressed? And how can we be sure the code is actually improved? Is there a way to measure those improvements, if there indeed are any? The solution S04 offers to these questions is Assured LLM-based Software Engineering.

Just like applications that utilize computer searching fall within the domain Search Based Software Engineering (SBSE), S04 proposes the term Large Language Model Based Software Engineering (LLMSE) should be used for applications that utilize LLMs in their functionality. The study further splits this area into online and offline LLMSE. This es-

sentially represents whether the results need to be real time, which is the case for online LLMSE. The definition for 'real time' S04 uses is that a result is real time if the entity that would receive the LLM response, a *consumer*, would not benefit in any way if the response from the LLM would be available faster. This means that how fast real time is depends on what the consumer is: a response that is real time for a human might not be so for another application. The third qualification for LLMSE is the one that interests us: LLMSE is said to be Assured if the LLM responses are paired with some verifiable claim or claims about the utility of the answers. This might also mean that the response itself has been post-processed to meet this standard. Being Assured is an independent property of LLMSE being offline or offline. This means that Assured LLMSE could be either online or offline in theory. In practice however, S04 posits that the computational requirements for Assuredness are most likely considerable enough, that Assured LLMSE errs on the side of offline LLMSE. Online LLMSE is still brought up by the study as an open research problem.

The assurances Assured LLMSE provides comes through filters, which mimic the fitness functions that are found in the Genetic Improvement approach. Instead of being purely boolean, these filters can also report the degree of failure for the result. How the eventual output is assured to meet these filters is achieved by a generate-and-test approach. Many more solutions are generated than is eventually provided for the consumer. This methodology also parallels Genetic Improvement approach closely. Indeed, the gradual advancement of the candidate solutions is done by applying the SBSE techniques on the prompts that are given to the LLM.

There are multiple code generation areas where Assured LLMSE can be utilized. S04 lists debugging, local optimization and global optimization as potential application fields. In local optimization, the LLM is given a limited context in which to perform the optimization, in other words, not the source code of the whole application. Global optimization is the opposite of this. Although S04 posits that global optimization is more likely to provide a meaningful result, it is also the more technically challenging of the two fields. Still, the field that S04 highlights as the best candidate for Assured LLMSE is refactoring. There are multiple aspects in which the field aligns well with assurance. Firstly, refactoring should not produce changes in the behavior of the software. This fact, in itself, can be used as a filter. More importantly, it is a filter that is easily automated. Secondly, refactoring is a field where there is virtually no need for the Assured LLMSE to be online. When considering these benefits, S04 finds the lack of exploration of LLMSE in the context of

refactoring surprising.

4.4.5 Targeted Verification Questions

Another angle in improving the code generation capabilities of LLMs is provided by S05: verification questions. The advantage of this approach is that the code generated by the LLM does not have to be run, unlike, for example, incorporating an error message into the prompt. The logic is that just like developers pose questions about their or their colleagues's solutions, the LLM would be asked about the potentially buggy parts of its code. This would require automatically locating the potentially buggy areas in code and generating a response based on that.

To aid in detecting the potential bugs, S05 suggest converting the generated code into an Abstract Syntax Tree (AST). It represents the syntactic structure of the code, but abstracts into a tree shape, removing unnecessary information like punctuation. AST categorizes parts of the code into different parts, such as name, arguments, where there is a function call, or what different attributes the components might have. This structure allows developers to pinpoint specific bug-patterns. This is what makes the verification questions targeted: they can be tailored to specific bugs that LLMs are more likely to create. S05 cites that LLMs often hallucinate objects and use wrong attributes. Both of these are patterns that are linked directly to some nodes in the AST representation of the code. While these are the types of bugs S05 repaired in their case study, AST supports the finding of many more bug-patterns.

By extracting the information from the potentially buggy codes, it is possible to create the targeted verification questions using templates. In S05, they additionally used another LLM to generate the question based on a formatting template. By iterating this process and chaining these verification questions together, the LLM should be able to fix the bugs the questions point to, thus improving the generated code. In their case study, S05 found that it reduced the number of bugs in the tracked categories (Hallucinated object and Attribute Error) by 40% to 62%, while introducing bugs to code that was functional only 12% of the time.

4.4.6 Metamorphic and Mutation-based Consistency Testing

S07 and S08 both deal with measuring LLMs code reading capabilities, and share an author. While this capacity is not directly useful in testing, many of the other solutions do rely on this capacity in some sense. More importantly, the way these studies reached their conclusion demonstrates additional ways to test LLMs.

S07 tests the code understanding capacity of LLMs by using metamorphic testing. It is a traditional software testing method that is used when the mechanisms that determine that a test should pass are hard to define, expensive to create or simply don't exist. This makes metamorphic testing quite a natural fit for LLMs. Instead of testing the unknown correct responses, the testers leverage metamorphic relations. These essentially define how the output of the software should or should not change based on the input. The example given is S07 is sorting. If the input array is reversed, the output should not change. This is a metamorphic relation. It is important to note, that this means that metamorphic testing requires multiple executions of the software with different inputs. S07 tests the LLM under study via two different changes. Does the understanding change if only the part of the prompt that describes the task at hand changes while the code remains unchanged, and does changing values or relational operators within the code impact the understanding. The metamorphic relation that is leveraged here is that if the LLM truly understands the code itself, value changes or differently worded tasks should not affect the outcome, since the understanding should be derived from the structure of the code. S07 found that these changes did affect the outcome, especially with logical operator changes. Although this result is not important to the subject of the review, the way they reached it could be utilized in other contexts, as long as there is some metamorphic relation to utilize.

We assume that S08 is a followup study to S07, since it tests the code understanding of LLMs when the logical operators in the code are changed, which they identified as a particular issue in S07. Instead of metamorphic testing, this time they use a novel method called Mutation-based Consistency Testing (MCT). MCT uses mutations in code, like in mutation analysis in traditional software testing, to create inconsistencies in the descriptions of code and what they actually do. The logic follows, that if the LLM is capable of noticing this dissonances, it should also notice the unintended inconsistencies in code. From this description, while it is not stated directly by S08, we consider MCT to be a subcategory of Metamorphic Testing. The relation between the code and the description is a metamorphic one: if the underlying code changes behavior and the description does not reflect this, the response should change.

Name	Description	Example
AOR	Arithmetic Operator Replacement	a + b → a - b
LVR	Literal Value Replacement	10 → 9
ROR	Relational Operator Replacement	a < b → a >= b
STD	Statement Deletion	remove one line

Figure 4.2: Some possible Mutation Operators

The mutation operators S8 tests the subject LLMs with are Arithmetic Operator Replacement(AOR), Literal Value Replacement(LVR), Relational Operator Replacement(ROR), and Statement Deletion(STD). Examples of how each of these operators change the code can be found in Figure 4.2. We recreated it directly from S08, since it illustrates the changes very clearly. It is important to remember that mutations can create identically functioning programs. In these cases the LLMs response should not change. Furthermore, S08 cites studies that prove that the problem of equivalent mutant detection is undecidable. This means that the mutants that are given to the LLM will require some human oversight.

Again, although S08 focuses on code reading capabilities of LLM and thus uses Mutation Operators for code, we feel that these ideas could be expanded and used in other context. Whenever two pieces of contents in a prompt have metamorphic relation, some mutation operator could be created that tests whether the LLM notices the discrepancy.

4.4.7 Data Slices

To address the issues regarding correctness and defining when exactly has an LLM-based system regressed, S13 suggests using Data Slices. Since looking at the regression of individual tests most likely proves to be intractable. On the other hand, simply looking at the entire data set would not help locating the issues. Additionally, regressions can happen even when the performance overall improves, thus looking at just the big picture won't even reveal that regession has happened if it is not uniform across tasks. The answer must lie somewhere in the middle. The preliminary results of S13 suggests that semantic slices could be used as the basis for regression test suites. By grouping the tests semantically, where the regressions happen can be localized. As an example what these groupings can be, S13 dealt with toxicity detection and they found that the regressions happened when dealing with toxic comments more than non-toxic. Within the toxic comments the re-

gression happened more often when the toxicity was triggered by politics, targets code or is just severe. All of these details would not have been found by just looking at the big picture, or the commonality between where regressions happen might have gone unnoticed if each test failed test was regarded just as an individual regression.

S13 does note that their data slices were created using metadata in their data set. Since such metadata is not available to all datasets, additional methods for finding these slices are needed. The study points to this area as an area of future reasearch and suggests exploring whether excisting approaches from the field of slice discovery and error analysis could be applicable here.

4.4.8 LLMs evaluating LLMs

Multiple of the reviewed studies cited LLMs themselves as a possible answer to their open questions (S01, S05, S11). We think the same could be possible for some of the areas of research suggested in this chapters by the reviewed papers: creating filters for Assured LLMSE, finding data slices for regression testing, or being the test oracles for the LangBite model. It does feel like using LLMs for testing LLMs, even if in differing ways, that one is just pushing the problem down the road. The results of S09 also point toward this direction. The study tested the quality of tests generated by a single LLM for the code it generated. Their result show that the generated tests where untrustworthy indicators for the correctness of the code they were testing, were low quality and had common test smells. This means that humans must remain part of the process when generating tests for LLM generated code, and as the authors succintly put, 'there is no free lunch'.

These results are discouraging when it comes to LLMs solving some issues with testing LLMs, but we feel that there is hope. Generating test code for generated code, is very close if not more difficult than the original task. The tasks where LLMs could help might be less complex than the task that is being tested. If that is the case, testing the performance of the LLM for that task might not have issues where LLMs are needed. Maybe the slices are more identifiable by humans, or the filters more clear. Although it is good to keep the results of S09 in mind, this should not be seen as definitive proof that the area of LLMs evaluating LLMs should not be explored further.

5 Discussion

In the first section of this chapter we will discuss the limitations and threats to the validity of this review. We feel it is important to discuss these potential issues before we present the final thoughts and directions for work in the future for this topic, which we will do in the final section of the chapter.

5.1 Limitations of the review

The elephant in the room for this systematic literature review is the sample sizes. A review of two studies would not provide much useful information, but we do not consider the expanded number of thirteen we reached much better. The steps outlined by Mäntylä in the guide we discussed in the Method chapter did not give numbers for how many studies in the final review would be too few, although the guide stated that the search step should produce between 20 and 400 results. The lower bound is that low because unlike traditional systematic literature reviews (Kitchenham and Charters, 2007), these reviews are conducted by one person. This is another flaw in the approach, although it was done knowingly. Having only one person to conduct the review increases the chance for mistakes to occur and personal biases to creep in. Due to the issue of low sample size, the latter concern is fortunately much less likely to occur.

Another way this review differs from the normal guidelines for systematic literature reviews is the lack of quality evaluation. This is another symptom of the low number of studies that remained after the inclusion and exclusion criteria. Potential issues with quality were considered when presenting the results, giving less weight to observations from studies that were deemed to have any problems.

The milder issues we were just small mistakes in grammar. The most notable case was a paragraph from S11, where 'the most significant question needed to answer' was 'therefore, are LLMs excel at causal reasoning?'. This does not resemble any traditional spelling mistakes we are aware of, and resembles in our opinion low quality machine translation. In theory, such mistakes should not affect the quality of the paper, but in practice it does raise more questions about the quality. Especially if such an egregious one is in the central question of the study.

The more major issues we encountered were in S09. The topic of the study, the capacity of LLMs to assess the reality of driving scenarios, was not important to the review, but the study did corroborate some observations found in other studies and the issue was not related to these, so was not excluded S09. The issue itself was with their research methodology. They only tested if the LLM could identify realistic scenarios as realistic, but not if they could also identify unrealistic scenarios as unrealistic. Although they acknowledged this problem in their study, we still consider a methodology that would describe a machine that responded 'Yes' to every scenario as perfectly capable of identifying the realism of scenarios problematic.

5.2 Future work and research

First of all, although we have stated the issue of very little research being done on the topic multiple times, it bears repeating once more. Any research done on the topic of software testing of LLM-based software or systems would benefit the field, no matter the angle. Even re-doing this review with a search string that targeted the type of studies that the expanded criterion included would most likely highlight issues and solutions that were not available to be reviewed with our search string. Nevertheless, there are avenues of future research we feel might be especially beneficial for the topic. We will discuss three of them in this section of the chapter: common vocalbulary and prompting related tools, and do simpler tasks for LLMs make testing them easier.

The low study count and the release dates of the papers do point to the testing of LLM-based software being a very new field for research. A symptom of this is lack of a standardized vocalbulary. Super LLMs, software interrogation, data slices, mutation-based consistency testing, verification questions; all of these terms were in single studies in the review. For some of these, this is by design. The authors were presenting a new term, but some of the new terms had overlapping ideas and definitions with each other. Concerns and filters, for example, were very similar in concept. We do not have a method for moving towards this more standardized vocalbulary. A good place to start might be for researchers to simply be more aware of what has already been written. If something someone has written matches the ideas your research is dealing with, utilize some of the words used in the previous work.

The importance of prompting was the most commonly found theme among the studies. Twelve out of thirteen studies mentioned prompt design as some part of their process. It

is a discipline unique to language models, so it is only natural that prompting and prompt engineering, the field of crafting better prompts, are involved when testing LLMs. Despite this importance, the tools for integrating prompting into IDEs and versioning them are lacking. This we feel is the most actionable course for future work. Chainforge took some steps, especially in making prompting more accessible to multiple fields, but the more programmer facing tools are not supporting the needs of LLM-based software.

Many of the open questions brought up in the studies were ones where LLMs themselves might be able to provide the answers. However, if the issue is testing whether an LLM works correctly, we are just passing the problem further down the road. Not all tasks are created equal however. If simpler tasks are simpler to test as correct, open questions which are simpler to solve or test as correct could still help with integrating LLMs for the more complex tasks. This is why we feel this is another worthwhile problem to focus future research on.

6 Conclusions

We conducted a systematic literature review of the testing of LLM-based software. Because the original criteria left us with only two studies, we expanded the topic to include anything related to LLMs from the software testing perspective, without changing the actual search string. This expansion left us with thirteen studies.

We discussed the relevant information the studies in the result section, which offered both solutions and issues to testing both specifically LLM integrations and LLM testing in general. We found few commonalities, which was not surprising to us due to the amount of papers the expanded search provided being on the low side for systematic literature reviews. We notes this fact and the some quality issues in some of the found papers as threats to the validity of the review.

We identified that further research was needed in general, since existing work on the topic was still so sparse. For the literature that exists, we found the need for common language to be a pressing issue, since many of the studies gave differing names for very similar concepts. How the oracles that are needed to test LLMs change in respect to the simplicity of the task is also an area with ample research opportunities. Most importantly, we identified the area where the most concrete action can be taken: prompting. The tools for prompting currently lose a lot of valuable information and very basic software production support, such as versioning, is quite lacking for them. Better tooling was also brought up as a potential solution for how multi-disciplinary working with LLMs currently is. If integrating prompts to LLM-based software would not require a background in software engineering, the pool for candidates could surely be expanded.

Bibliography

22 Top AI Statistics And Trends In 2025 (2024). URL: https://www.forbes.com/advisor/business/ai-statistics/ (visited on 02/04/2025).

About Google Scholar (2025). URL: https://scholar.google.com/intl/fi/scholar/about.html (visited on 01/29/2025).

Bentley, J. E. (2005). *Software Testing Fundamentals—Concepts, Roles, and Terminology*. URL: https://support.sas.com/resources/papers/proceedings/proceedings/sugi30/141-30.pdf (visited on 02/24/2025).

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). *Language Models are Few-Shot Learners*. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165.

ICST 2024 (2024). URL: https://conf.researchr.org/home/icst-2024 (visited on 02/06/2025).

Jabborov, A., Kharlamova, A., Kholmatova, Z., Kruglov, A., Kruglov, V., and Succi, G. (2023). "Taxonomy of Quality Assessment for Intelligent Software Systems: A Systematic Literature Review". In: *IEEE Access* 11, pp. 130491–130507. DOI: 10.1109/ACCESS.2023.3333920.

Kästner, C. (2025). *Machine Learning in Production: From Models to Product*. URL: https://mlip-cmu.github.io/book/09-quality-attributes-of-ml-components.html (visited on 02/24/2025).

Kitchenham, B. A. and Charters, S. (July 2007). *Guidelines for performing Systematic Literature Reviews in Software Engineering*. English. Tech. rep. EBSE 2007-001. Keele University and Durham University Joint Report. URL: https://www.elsevier.com/__data/promis_misc/525444systematicreviewsguide.pdf.

Laakso, A. (2023). *Ethical challenges of large language models-a systematic literature review*.

Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., and Neubig, G. (2021). *Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing*. arXiv: 2107.13586 [cs.CL]. URL: https://arxiv.org/abs/2107.13586.

Luitse, D. and Denkena, W. (2021). "The great Transformer: Examining the role of large language models in the political economy of AI". In: *Big Data & Society* 8.2, p. 20539517211047734. DOI: 10.1177/20539517211047734. eprint: https://doi.org/10.1177/20539517211047734. URL: https://doi.org/10.1177/20539517211047734.

Mini Systematic Review Guide for Masters's thesis (2023). URL: https://mmantyla.github.io/mini-slr-for-thesis/ (visited on 01/29/2025).

Oh, Air Canada! Airline pays out after AI accident (2024). URL: https://techhq.com/2024/02/air-canada-refund-for-customer-who-used-chatbot/ (visited on 02/19/2025).

Radford, A. and Narasimhan, K. (2018). "Improving Language Understanding by Generative Pre-Training". In: URL: https://api.semanticscholar.org/CorpusID:49313245.

Appendix A Reviewed studies

Table A.1: Studies used

Id	Initial result	Authors	Release date	Title
S01	Out	Morales S.; Clarisó R; Cabot J.	2024-09-24	A DSL for Testing LLMs for Fairness and Bias
S02	Out	Porter A.; Karr A.	2024-09-17	Active Model Learning for Software Interrogation
S03	Out	Ouyang S.; Zhang J.; Harman M.; Wang M.	2024-09-26	An Empirical Study of the Nondeterminism of ChatGPT in Code Generation
S04	Out	Alshahwan N.; Harman M.; Harper I.; Marginean A.; Sengupta S.; Wang E.	2024-04-14	Assured LLM-Based Software Engineering
S05	Out	Ngassom S.; Dakhel A.; Tambon F.; Khomh F.	2024-09-10	Chain of Targeted Verification Questions to Improve the Reliability of Code Generated by LLMs
S06	Out	Arawjo I.; Swoopes C.; Vaithlingam P.; Wattenberg M.; Glassman E.	2024-05-11	ChainForge: A Visual Toolkit for Prompt Engineering and LLM Hypothesis Testing
S07	Out	Li, Ziyu; Li, Zendu; Xiao K.; Li, Xuan	2024-02-15	Evaluating LLM's Code Reading Abilities in Big Data Contexts using Metamorphic Testing
S08	Out	Li, Ziyu; Shin D.	2024-06-18	Mutation-based Consistency Testing for Evaluating the Code Understanding Capability of LLMs
S09	Out	Zilberman S.; Cheng B.;	2024-09-17	"No Free Lunch" when using Large Language Models to Verify Self-Generated Programs
S10	Out	Wu J.; Lu C.; Arrieta A.; Yue T.; Ali S.	2024-07-30	Reality Bites: Assessing the Realism of Driving Scenarios with Large Language Models

Id	Initial result	Authors	Release date	Title
S11	In	Barnett S.; Kurniawan S.; Thudumu S.; Brannely Z.; Abdelrazek M.	2024-06-18	Seven Failure Points When Engineering a Retrieval Augmented Generation System
S12	Out	Li, Shun-Hang; Zhou G.; Li, Zhi-Bo; Lu, Ji-Cang; Huang, Ning-Bo	2023-12-25	The Causal Reasoning Ability of Open Large Language Model: A Comprehensive and Exemplary Functional Testing
S13	Out	Ma, W; Yang C.; Kästner C.	2024-07-18	(Why) Is My Prompt Getting Worse? Rethinking Regression Testing for Evolving LLM APIs