UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Karel Udras

Automated Resolution of Issue Reports using LLM Agents: A Systematic Literature Review

Bachelor's Thesis (9 ECTS)

Supervisor: Faiz Ali Shah, PhD

Tartu 2025

Automated Resolution of Issue Reports using LLM Agents: A Systematic Literature Review

Abstract: The goal of this thesis is to evaluate the ability of large language model agents in resolving issue reports present in repositories. A systematic literature review following Kitchenham's methodology is conducted. Systematic literature review covers studies published in 2023 and later. Articles for this systematic literature review were selected according to their relevance towards large language model agents designed for issue resolution. The systematic literature review identifies common agent designs, evaluation datasets and benchmarks, metrics, and factors influencing their performance. Findings from this thesis highlight the strengths and limitations of current large language model agents and their benchmarks.

Keywords:

Large language model, bug repairing, large language models agents, benchmarks

CERCS: P176, Artificial intelligence

Automatiseeritud vearaportite lahendamine LLM agentide abil: süstemaatiline kirjanduse ülevaade

Lühikokkuvõte: Töö eesmärgiks on hinnata suurte keelemudelite agentide võimekust tarkvaravigade automaatsel parandusel, keskendudes nende võimekusele lahendada vearaporteid koodihoidlates. Töö käigus viidi läbi süstemaatiline kirjanduse ülevaade vastavalt Kitchenhami metoodikale, mis hõlmas 2023. aastast alates avaldatud uurimistöid. Artiklid süstemaatilise kirjanduse ülevaate jaoks valiti vastavalt asjakohasusele suure keelemudeli agentide suunas, mis on mõeldud veakirjalduste lahendamiseks. Süstemaatiline kirjanduse ülevaade leidis, millised on tüüpilised agentide disainid, andmestikud ja mõõtlusalused, näitajad ja faktorid nende tulemuslikkust mõjutasid. Selle lõputöö tulemused toovad esile praeguste suurte keelemudeli agentide ja nende mõõtlusaluste tugevused ja piirangud.

Võtmesõnad:

Suur keelemudel, programmivigade parandus, suure keelemudeli agendid, mõõtlusalused

CERCS: P176, Tehisintellekt

Table of Contents

Terms and Definitions

Large Language Model (LLM) A category of foundation models trained on immense amounts of data making them capable of understanding and generating natural language and other types of content to perform a wide range of tasks[1].

Transformer Architecture A deep learning model architecture based entirely on attention mechanisms, dispensing with recurrence and convolutions entirely, to model the relationships between elements in a sequence[2].

Large Language Model Agent AI systems that leverage Large Language Models (LLMs), tools, and memory to perform tasks, make decisions, and interact with users or other systems autonomously[3].

Zero-Shot Learning A technique that enables pre-trained models to predict class labels of previously unknown data, i.e., data samples not present in the training data[4].

Few-Shot Learning A machine learning technique that enables models to learn from a limited number of labeled examples, typically fewer than 100, and generalize well to new, unseen data. This approach is particularly useful when there is a lack of large-scale labeled datasets or when the data is expensive to collect or label[5].

Integrated Development Environment (IDE) A software application that helps programmers develop software code efficiently. It increases developer productivity by combining capabilities such as software editing, building, testing, and packaging in an easy-to-use application[6].

Code Review A code review is a peer review of code that helps developers ensure or improve the code quality before they merge and ship it[7].

---

[1] https://www.ibm.com/think/topics/large-language-models

[2] https://arxiv.org/abs/1706.03762

[3] https://www.k2view.com/what-are-llm-agents/

[4] https://encord.com/blog/zero-shot-learning-explained/

[5] https://shieldbase.ai/glossary/few-shot-learning

[6] https://aws.amazon.com/what-is/ide/

[7] https://about.gitlab.com/topics/version-control/what-is-code-review/

In-Context Learning A technique where task examples are integrated into the prompt in natural language format. This approach lets pretrained large language models solve new tasks without fine-tuning the model[8].

RAG is a technique for enhancing the accuracy and reliability of generative AI models with information fetched from specific and relevant data source[9].

[8] https://www.lakera.ai/blog/what-is-in-context-learning

[9] https://blogs.nvidia.com/blog/what-is-retrieval-augmented-generation/

Introduction

Resolving software bugs is an inevitable part of software development. In code repositories, the bugs are pointed out by issue reports containing a description of the problem. Software bugs can significantly affect a software product's usability, maintainability, and reliability. In large collaborative environments, issue reports accumulate rapidly, making manual bug resolution a time-consuming and resource-intensive task. Efficiently identifying and fixing bugs and resolving issue reports remain major challenges in software engineering. The topic of LLM agents for GitHub issue fixing is a relatively new topic, the first articles about the topic started emerging in 2023 as the idea of benchmarking that was proposed and benchmarks were created by Jimenez et al [4].

Recent advances in large language models (LLMs) have improved the ability of LLMs to understand and generate natural language [1]. LLMs are increasingly being implemented into the following programming tasks: code generation, error detection, and automated debugging [2]. Their understanding of natural language and source code allows them to interpret issue descriptions, analyze code context, and suggest or even implement fixes.

LLM agents combine these LLMs with additional components such as tool integration, planning mechanisms, or memory, allowing LLM agents to perform complex, goal-driven tasks [3]. Such LLM Agents can analyze GitHub issue reports, identify relevant code sections, and attempt to fix bugs through iterative interactions with the codebase. Research has shown that LLM-based agents can assist human developers in activities such as code review, bug classification, and test case generation [3], [4].

Although LLM agents are providing promising results, the LLM agents are not widely used in real-world debugging scenarios. Existing approaches of LLM agents and their benchmarks differ from each other in design, evaluation methods, and application domains. Also, the field is still emerging, and the first agents and benchmarking tools started appearing as of late. This thesis conducts a systematic literature review following Kitchenham’s guidelines [5] to gather and analyze recent research on LLM-based agents used for resolving repository-level issues. This thesis systematically reviews the literature that assesses the performance of these agents.

The goal of this thesis is to examine what types of LLM agents exist, how do the LLM agents perform, what kind of benchmarks exist for them, what factors influence their performance and the consistency of their results across different datasets. By conducting a

structured analysis of the latest research, this thesis aims to provide insight into the practical strengths and limitations of LLM agents for automated bug fixing.

The first chapter is about background information on LLMs and LLM agents. In the second chapter is background information about Systematic Literature Review (SLR). In the third chapter is related work to this thesis. In the fourth chapter the methodology of the SLR is explained. In the fifth chapter the results are covered for the research questions. In the sixth chapter is discussion about current state of LLM agents and their future.

1. Large Language Models

This thesis focuses on large language models (LLMs) that are instruction-tuned with reinforcement learning from human feedback (RLHF). LLMs are artificial intelligence (AI) based computational models, their natural language processing (NLP) evolved from statistical to neural language modelling and then from pre-trained language models (PLMs) to LLMs [6]. Compared to current models, early large language models were very rudimentary. Today's models can, for instance, execute code, something that early models could not do [7].

LLMs are designed to understand and generate human-like text [8]. These models have a very strong command of natural language enabling them to handle complex tasks, allowing them to better understand people and more accurately gauge the user's actual intentions [1]. Consequently, large language models can produce answers that are more coherent and understandable, and they also exhibit strong capabilities in generalizing and providing arguments in response to prompts, thus being more prepared for unexpected tasks [1]. Typically, "large language model" refers to transformer-architecture-based language models consisting of hundreds of billions of parameters, trained on massive text datasets [1].

Those models are increasingly assisting us in daily tasks. Their ability to solve problems and provide recommendations improves every day. As their capabilities grow, large language models are employed in ever more domains. Many companies are investing in or developing these models, accelerating their evolution. LLMs play an innovative role across various industries, for example disease diagnostics, automotive industry, consumer behavior analyses, fraud detection, customer service automation and risk management [8]

More advanced techniques like in-context learning can boost a language model's capabilities without requiring downstream tasks to be fine-tuned [1]. Furthermore, by empowering the model to strategically use prompts (for example, through chain-of-thought prompting), the model can generate output step by step and thereby handle more complex decision-making processes [1]. LLMs have advanced abilities (such as reasoning, planning, decision-making, in-context learning, etc) and are widely used in several settings like robotics, tool manipulation, question answering, autonomous agents and other settings. [6]. With these powerful features, language models demonstrate great potential for providing recommendations [1].

It is important to understand each model's potential and how well each one solves problems or gives suggestions. Some models excel at writing code, while others handle everyday problems better. Overall, language models can solve complex problems or provide useful suggestions to help users solve a problem. There are several LLM-based technologies used to address the challenge of software bugs. Zubair et al. [9] outline several approaches in their recent paper [9], including: LLMs that suggest potential fixes, models fine-tuned through strategies such as "Knowledge-Intensified fine-tuning" and "Repair-Oriented fine-tuning," LLM-based automated program repair (APR) systems, frameworks that support continual learning of bug fixes across multiple programming languages, methods designed to handle complex software defects involving interdependent code changes, and a curricular fine-tuning technique aimed at enhancing LLM effectiveness in APR tasks.

The study of Feng et al (2024) [10] reads that most LLM-based repair approaches underutilize or use limited software artifacts. The authors assume that LLM-based program repair tools would be more effective if they could use software artifacts. By providing LLMs with a rich set of contextual data, they could understand and localize bugs more accurately and offer more precise fixes.

Despite their strong potential, the use of LLMs comes with several challenges. These models often require lengthy training and inference times, demand significant hardware resources, and incur high operational costs [6]. Additionally, Raza et al. [8] highlight ethical concerns and biases present in the training data.

1.1 LLM agents

As large language models advance, so do their agents, because an agent's capabilities are largely determined by the underlying model it uses but also by how the agent itself is constructed.

Large language model agents are systems that leverage language models to interact with tools that are usually implemented into integrated development environments (IDE) [2]. The interaction between the LLM and the tools can be seen in Figure 1, which is inspired by SuperAnnotate's blog about LLM agents [11]. These agents can write code, find and fix bugs, and support software-system planning [2]. Such solutions have a significant impact on the bug-fixing of publicly accessible code repositories [2]. Because large language models have a relatively robust understanding of natural language, these agents can

effectively interpret source code, break it down into relevant parts and incorporate user-provided descriptions of the issue and any code comments [2].

Furthermore, large language model agents can search for essential information in code repositories, make changes to the code, run the code, and verify the correctness of their modifications [2]. It has also been stated that large language model agents can significantly streamline the traditional code review process, increasing code quality and aiding a developer's professional growth by pointing out errors in the developer's code [3]. The agent's ability to detect a large variety of deviations from coding best practices, to recommend code optimizations, and to apply high-quality coding standards is a remarkable step toward automating processes and maintaining code quality [3]. Positive feedback from software developers about large language model agents sets a precedent that they could be used for finding code errors [3]. Therefore, it is important to assess how effectively these agents can fix software bugs and, if the results are good, to see whether they can prevent critical errors that might go unnoticed by human eyes. Additionally, it is vital to know whether the agent can repair software errors at the project, function, or individual line-of-code level.

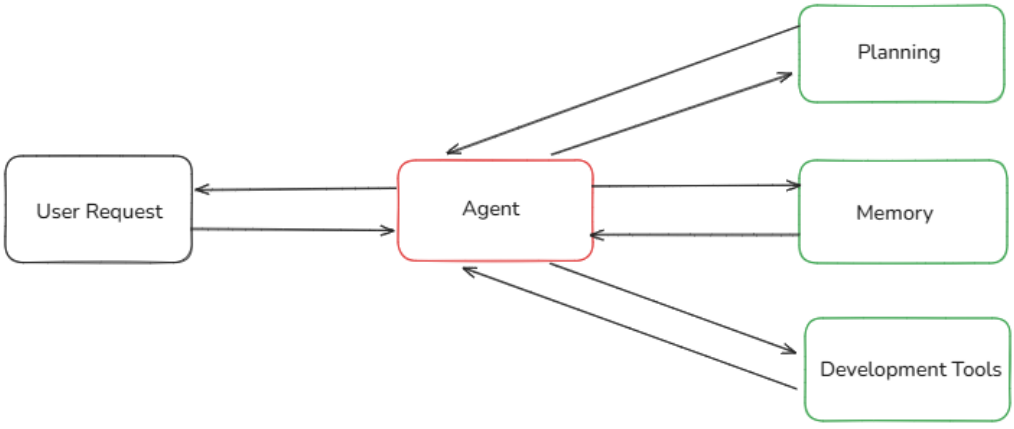


Figure 1. LLM agent's workflow (inspired by [11])

1.2 Using Prompts in Large Language Models

The core idea behind using prompts is that the model's parameters should not be updated. Instead, one should adapt the large language model for downstream tasks by using the correct prompts [1]. At earlier stages, traditional prompting methods were used, focusing

on integrating downstream tasks into a single textual output for the end user, such as text summarization, extracting relationships, or sentiment analysis [1]. After that, in-context learning has emerged as a powerful prompting technique that allows large language models to learn new tasks based on contextual information provided in the prompt [1].

Traditional prompting generally follows two main approaches [1]. The first is prompt engineering, which creates prompts by imitating text that the language model is already familiar with from training [1]. This methodology enables the trained model to merge downstream tasks with unexpected tasks into a single language-generation objective, unifying them under a shared goal [1]. The second main approach is providing the model with incremental prompts, which include a few examples of both input and output [1]. This technique allows the model to respond appropriately to various downstream tasks [1]. Since there is a significant gap between natural language generation tasks and downstream recommendation tasks, most traditional prompting methods have succeeded only in narrower use cases [1].

In-context learning is considered an advanced prompting technique that makes the language model far more efficient in tackling a variety of downstream tasks [1]. The success of in-context learning is largely attributed to two main elements: few-shot learning and zero-shot learning [1]. Generally, few-shot learning outperforms zero-shot learning because the model is given additional contextual examples [1]. There are different kinds of prompting techniques which use the concept of in-context learning, to better utilize the capabilities of LLMs in certain scenarios. One of them being Chain-of-Thought Prompting (CoT), which is used when the LLM can benefit from being guided through intermediate reasoning steps [12]. Another one is a tree-based reasoning structure Tree-of-Thoughts (ToT), which is usually worth implementing when enabling LLM to explore numerous reasoning paths while evaluating those iteratively [13]. Consequently, it is essential to understand which kind of prompting techniques to use and in what form and how the choice of prompt affects the model's response.

2. Systematic Literature Review

A systematic literature review is a comprehensive and replicable process for identifying, evaluating, and synthesizing existing research relevant to a certain research question or area of interest. Unlike traditional narrative reviews, which are sometimes subjective and non-replicable, a systematic literature review adheres to a well defined methodology to ensure transparency, reduce bias, and improve the reliability of findings [5], [14].

The use of systematic literature reviews in computer science and software engineering has significantly risen, particularly in areas that involve rapid innovation and evolving approaches, such as artificial intelligence and machine learning. In software engineering, Kitchenham [5] developed a formal framework for the application of systematic review principles. This framework emphasizes the importance of predetermined inclusion and exclusion criteria, rigorous quality evaluation of sources, and exhaustive database searches.

Systematic literature reviews are particularly beneficial in emerging research domains such as large language models (LLMs) and autonomous agents, when scholarly work is dispersed across multiple disciplines and publication platforms. This thesis examines how LLM agents can aid in resolving GitHub issues, and a systematic literature review provides a methodical analysis of current strategies for issue resolution, automation in software repositories, and the incorporation of LLMs into development workflows.

This thesis is anchored in a comprehensive literature assessment, ensuring that its contributions are informed by existing knowledge, prevent redundancy, and conform to established norms in empirical software engineering research [5], [15].

3. Related work

The use of LLMs in the context of software issue resolution has recently been analysed by Zubair, Al-Hitmi and Catal in 2024 [9], where the results of 41 studies were analysed. In these studies most commonly Encoder-Decoder architecture and open-access datasets had been used. The results show that the effectiveness of different LLMs to repair the programs is very different. Authors suggest the combined use of different LLM architectures or LLM techniques combined with traditional program repair techniques.

In the article of "Multimodal LLM Agents: Exploring LLM interactions in Software, Web and Operating Systems" by Allen Thomas, Kartik Ramesh and Shraddhaa Mohan [16], the authors investigated different kinds of Multimodal LLM agents. Also they briefly mention fixing GitHub issues with a LLM agents but LLM agents for issue resolution were not the main focus of the paper and no analysis was conducted with gathered data.

4. Methodology

This thesis follows the systematic literature review methodology proposed by Kitchenham [5], which is widely adopted in software engineering. The review aims to identify, evaluate, and synthesize existing literature on large language model (LLM) agents developed to automate the resolution of repository issues. The systematic literature review process consists of three main stages: planning, conducting, and reporting.

4.1 Research Questions

The review is guided by the following research questions:

- RQ1: What benchmarks and metrics are commonly used to evaluate the performance of LLM-based agents?
- RQ2: What types of LLM-based agents are designed to automate the resolution of issue reports, and how do they perform?
- RQ3: What factors influence the performance of LLM-based agents in automating the resolution of issue reports?
- RQ4: Is the performance of LLM-based agents consistent across evaluations on multiple datasets?

RQ1 focuses on identifying what kind of benchmarks exist and which are commonly used to evaluate the performance of the agentic frameworks. RQ2 focuses on what agents are there for resolution of issue reports and how do they perform on current datasets. RQ3 focuses on what kind of factors influence the performance of the agentic frameworks currently available and whether there are similar factors used in multiple agents. RQ4 focuses on whether the results are consistent across multiple datasets also, it gives an idea how much does it matter on which dataset an agent was evaluated on.

4.2 Search Strategy

To locate relevant literature, a structured search was carried out using the following search string:

Search query: ("llm agent" AND "issue resolution") OR ("swe-bench" AND "llm agent")

Since this thesis is about llm agents and their capabilities about resolving issues on repository level. Search strings with writing out “large language models” instead of “llm”

was conducted but it produced fewer results, meaning that most papers either used both terms in their abstract or just LLM. Also it was tested out whether adding “GitHub issues” instead of “issue resolution” would produce more papers but the term for the quality and quantity of papers retrieved turned out to be the one used.

The second part of the search query used term “swe-bench” since it became obvious from looking into abstracts of papers from testing out different queries that most papers that have done experiments on issue resolving capabilities have included SWE-Bench datasets, so adding it to the search query was important to find missing papers about large language models issue resolving capabilities. The term “llm agent” was also kept because it is the focus of the study. And since the SWE-Bench was mentioned in the query it produces results that are about large language model’s agents that are built for issue resolution. Another reasoning behind the addition of “swe-bench” to the query is that the author has prior knowledge in the large language model agents and realised that articles that would be important to the study were not present.

The search focused on literature published in 2023 or later, as most research in this area emerged recently after the LLM models like GPT-3.5 and Claude-2 came to the market [4]. Only papers published in English were considered.

The search was conducted in the following academic and open-access databases:

- Google Scholar
- IEEE

The search query found 201 papers from Google Scholar and 0 results from IEEE.

4.3 Selection criteria for studies

The relevance and quality of the articles were ensured by applying predefined inclusion and exclusion criteria (Table 1).

The selection process involved four steps:

1. **Identification** of studies found from databases.
2. **Screening** of title and abstract to exclude clearly irrelevant studies and removal of duplicates.
3. **Eligibility** assessment of articles, consisting of full-text assessments.

4. **Determining included papers** using the inclusion and exclusion criteria to finalize the list of eligible studies.

Table 1. Inclusion and exclusion criteria of the search studies

Criteria Identifier	Criteria	Criteria reasoning
IC1	The paper must include benchmarking against a dataset that's used for evaluating LLM agents.	In order to discuss the performance of the LLM agents, benchmarking has had to be done to compare the results.
IC2	The agents used in the paper must focus on LLM agents that have a capability to resolve GitHub issues.	Some agents aren't built to be implemented into a pipeline for issue resolution.
EC1	Is the paper not written in English?	The paper has to be understandable.
EC2	Is the study published before 2023?	Studies about LLM agents focusing on issue resolution started appearing after 2023.

As visualized in Figure 2 which is a representation of a PRISMA Model [17], the first step was to identify articles that could be added to the selection process, in total 201 articles were identified. Then during the screening process duplicates were removed from the results of the search query. Since articles could only be found from Google Scholar, and after checking for possible duplicates, none were removed. Also 167 articles were excluded from the pool of studies. After conducting the eligibility assessment, out of the 34 articles remaining 9 were excluded after full-text review. The last step in the process was to determine papers that were included to the list of primary studies, which ended up being 25 studies. The list of those studies can be found from the Appendix II.

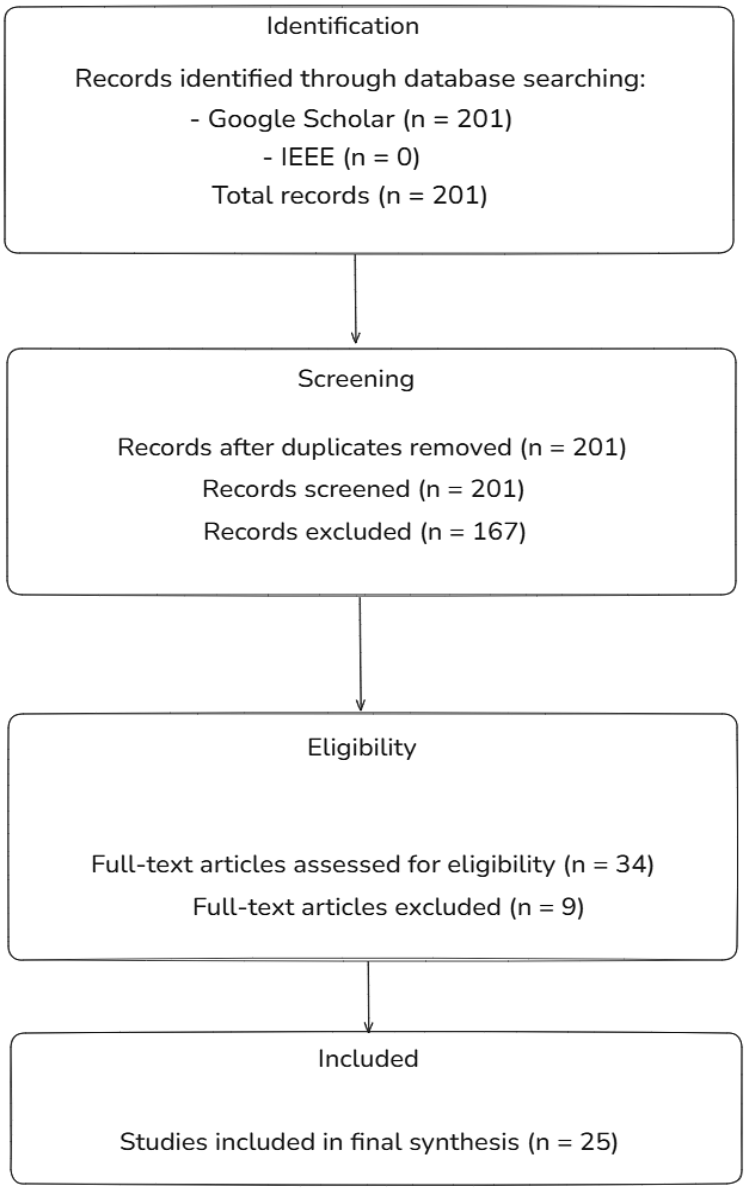


Figure 2. PRISMA model about selecting articles

4.4 Data Extraction

Relevant information from each included study was extracted using a structured data form. The extracted metadata and study details were used to answer each research question. The table below summarizes the data points captured.

Table 2. Data extraction form

Data about the source (metadata)	
Data about the page	Description
Identifier	Unique id
Title	Title of the paper
Authors	Names of the authors of the paper
University or organization	University or organization of researchers
Country	In what country is the university or organization located
Year	Year that the paper was published
Link	Link to the page of the paper
Date accessed	Date when the paper was accessed
Data supporting research questions	
Used dataset(s)	Which datasets were used for the experiments (RQ1)
Agents used	Which agent's were mentioned in the study for experiments and comparisons (RQ2)
Issue resolution rate	Percentage of resolved issues of the dataset (RQ2)
Factors influencing performance of agents	Factors that influenced performance of the agents (RQ3)
Agent's relative performance on multiple datasets	Were the resolve rates consistent on multiple datasets (RQ4)

This structured data collection ensures traceability and supports the transparent synthesis of findings presented in the next sections. The metadata about the papers was added to the spreadsheet "Data extraction" form, and then the data was extracted in accordance with Table 2.

4.5 AI Usage

In composing the thesis, artificial intelligence ChatGPT[10] was used for gathering ideas and finding inspiration, for author to develop their own ideas in order to write the text related to the motivation of the thesis in the introduction section and for paraphrasing across the thesis. For detecting grammatical mistakes QuillBot[11] was used.

[10] https://openai.com/index/chatgpt/

[11] https://quillbot.com/grammar-check

5. Results

This section answers four research questions (RQ1 to RQ4) defined in Section 4.1.

Metadata was collected during the SLR and heatmap about countries from where contributions were made from towards primary studies was created, which can be seen in Appendix I. The most contributions came from USA companies or universities, which was 10. After that came China with 5 contributions, Singapore with 4, Canada with 3, India with 2 and all other countries having a single contribution.

5.1 Existing benchmarks and their metrics (RQ1)

There are already many datasets for benchmarking large language model agents, as can be seen in Table 3, but from conducting the systematic literature review, it became clear which datasets were used the most for benchmarking large language models.

Table 3. Benchmark datasets used for the evaluation of LLM agents

Dataset	Count	Number of issue instances
SWE-Bench Lite [P2], [P5], [P6], [P7], [P10], [P11], [P12], [P13], [P14], [P15], [P16], [P18], [P19], [P20], [P21]	14	300
SWE-Bench [P1], [P15], [P17], [P18]	4	2294
SWE-Bench Verified [P5], [P20], [P23]	3	500
CrossCodeEval [P7], [P12]	2	Not specified
CSR-Bench [P4]	1	Not specified
DCA-Bench [P9]	1	91
Defects4J v1.2 [P5]	1	391
Defects4J v2 [P5]	1	835
EvoCodeBench [P7]	1	275
LocBench [P3]	1	660
RepoExec [P5]	1	Not specified
SWA-Bench [P17]	1	Not specified
SWEE-Bench [P17]	1	Not specified
SWE-Bench+ [P8]	1	2294
SWT-bench-lite [P24]	1	276
TDD-Bench Verified [P22]	1	449
Alibaba In-house dataset [P2]	1	194
Google In-house dataset [P25]	1	80

As seen on Figure 3, the SWE-Bench Lite consisting of was used in 14 papers, SWE-Bench full dataset was used in 4 papers, SWE-Bench Verified was used in 3 papers and CrossCodeEval was used in 2 studies. All other benchmarks were used only once in the papers that were included in the review process.

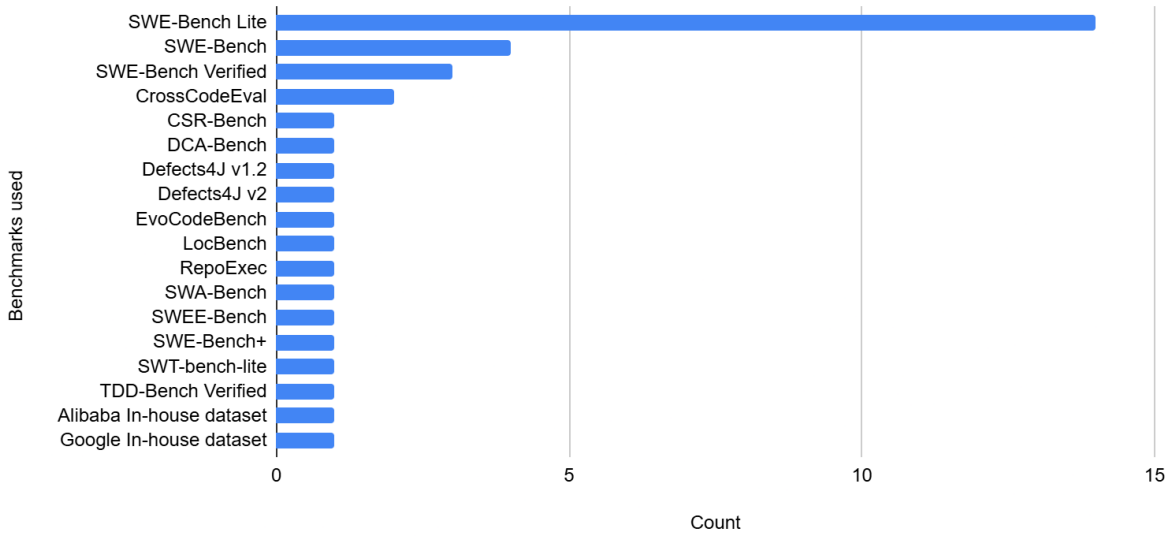


Figure 3. Datasets used in the selected research papers

The most commonly used metric for evaluating LLM agents for fixing GitHub issues is issue resolve rate. Other metrics are not used across different datasets.

Out of the SWE-Bench benchmarks the SWE-Bench Lite is the one with the least amount of GitHub issues, consisting of 300 instances to be resolved [4]. The next most popular dataset is SWE-Bench full dataset which consists of 2294 unresolved GitHub issues [4], since most agents are built to work on APIs of large language models, it's obvious why it's less popular than similar datasets containing fewer issues. The third most popular dataset is SWE-Bench Verified which consists of 500 GitHub issues and those have been also human validated for quality[4]. The popularity of SWE-Bench datasets could be contributed to the second part of the search query, which had "swe-bench" included in it. On the other hand, since the SWE-Bench was the first benchmarking tool for benchmarking large language models agent's it would be highly likely that it would be the most used one.

SWE-Bench+ pointed out that data leakage could be a major issue in the SWE-Bench datasets, meaning that over 94% of issues were available before the training cut-off dates of the large language models used [P8]. SWA-Bench, SWEE-Bench and SWE-Bench+ were only used in one study each and since the resolve rates are 2-4x worse than resolve rates on SWE-Bench [P18].

Results on datasets like Google's or Alibaba's can't really be compared to others because they have their own in-house datasets containing problems that were encountered developing their products. It's stated in Google's article that they have 80 issues of encountered bugs [P25]. Alibaba's in-house dataset contains 194 repository level issue instances [P2].

5.2 LLM agents and their performances (RQ2)

In total in the papers included in the systematic literature review, there were 32 different agentic frameworks mentioned and 60 different combinations with agents and large language models. Only the best combinations of agents and LLMs were mentioned in Table 4 and Table 5.

Most papers described testing their agent on a single dataset. Some papers have tested their agents on multiple datasets but papers that test multiple agents in order to compare their agent are not existing currently. Only exception being papers about adding another tool to the agent so they tested it out on multiple agents. If agents have not been tested on same dataset then a fair comparison can not be made. As seen in Table 4 and Table 5, the most experiments were ran on those datasets. SWE-Bench Lite consists of 300 issues and SWE-Bench Verified consists of 500 issues.

Data for Table 4 and Table 5 was gathered from tables of the primary studies, where they compared their agents performance against other agents. This was done because the agents performance would be relative, if they were tested on different datasets. Datasets that had the most results were SWE-Bench Lite and SWE-Bench Verified, so looking at agents' performance on those datasets made the most sense. On both of the tables, 25 best performing LLM agent and LLM combinations were listed, since that was the number of different results listed in primary studies for the SWE-Bench Verified. The resolution rates vary in Table 4 from 62.8% to 28.8% within the 25 results.

Table 4. LLM agents and their performance (i.e., Issue resolution rate) on SWE-Bench Verified

LLM Agent and LLM (if available)	Issue Resolution Rate
Blackbox AI Agent [P20]	62.80%
CodeStory Midwit Agent + SWE-Search [P20]	62.70%
Anthropic Tools + Claude3.7-Sonnet [P23]	62.30%
OpenAI Tools + OpenAI-o3-mini [P23]	61.00%
OpenAI Tools + Claude3.5-Sonnet-v2 [P23]	61.00%
Learn-by-Interact [P20]	60.20%
devlo [P20]	58.20%
Emergent E1 [P20]	57.20%
Gru (Claude 3.5 Sonnet) [P20]	57.00%
EPAM AI/Run Developer Agent [P20]	55.40%
Amazon Q Developer Agent [P20]	55.00%
PatchPilot (Claude 3.5 Sonnet) [P20]	53.60%
OpenHands + CodeAct v2.1 [P20]	53.00%
Google Jules + Gemini 2.0 F [P20]	52.00%
Engine Labs [P20]	51.80%
Agentless + Claude-3.5-Sonnet [P20]	51.00%
Agentless + OpenAI-o3-mini [P23]	51.00 %
Solver [P20]	50.00%
Bytedance MarsCode Agent [P20]	50.00%
SWE-SynInfer + SWE-Reasoner 32B [P23]	46.00%
Agentless + OpenAI-o1-1217 [P23]	41.00%
Agentless Mini + Llama3-SWE-RL 70B [P23]	41.00%
NebiusAI + NebiusAI 72B/8ToR [P23]	40.60%
SWE-SynInfer + Claude3.5-Sonnet-v1 [P23]	37.00%
SWE-agent + Claude3.5-Sonnet-v1 [P23]	36.00%
Agentless + GPT-4o [P23]	35.80%
SWE-Agent + Claude 3.5 Sonnet [P5]	33.60%
Agentless + GPT-4o [P5]	33.20%
HYPERAGENT-Full-1 [P5]	33.00%
SWE-SynInfer + GPT-4o [P23]	31.80%
HYPERAGENT-Full-2 [P5]	31.48%
HYPERAGENT-Lite-1 [P5]	30.20%
SWE-Gym + SoRFT-Qwen 32B [P23]	30.00%
SWE-Gym + SWE-Gym 32B [P23]	29.80%
AutoCodeRover + GPT-4o [P5], [P23]	28.80%

Agents that had the best performance on the SWE-Bench Verified dataset, which was the dataset that had second-most results can be seen on Table 4. Best performing agents were Blackbox AI Agent having 62.8% issue resolution rate [P20], CodeStory Midwit Agent + SWE-Search which had 48.67% issue resolve rate [P20] and Anthropic Tools + Claude3.7-Sonnet which had 48.33% issue resolve rate [P23]. It's important to mention that all of these agentic-frameworks have closed-source components.

Best results from a open-source agentic frameworks on the SWE-Bench Verified dataset were from PatchPilot having an issue resolve rate of 53.60%[P21], OpenHands + CodeAct v2.1 having an issue resolve rate of 53.00% [P20], Agentless + Claude-3.5-Sonnet having an issue resolve rate of 51.00%[P20] and Agentless + OpenAI-o3-mini having an issue resolve rate of 51.00% [P23].

Agents that had the best performance on the SWE-Bench Lite dataset, which was the dataset that was most experimented on the results can be seen in Table 5. Best performing Blackbox AI Agent having 49.00% issue resolve rate [P14], [P20], Gru which had 48.67% issue resolve rate [P14], [P20] and Globant Code Fixer which had 48.33% issue resolve rate [P14], [P20]. It's important to mention that all of these agentic-frameworks have closed-source components. The worst performance was on the SWE-Bench Lite dataset was 2.67% issue resolution rate by RAG + GPT-4 [P12].

Best results from a open-source agentic frameworks on the SWE-Bench lite dataset were from PatchPilot having an issue resolve rate of 45.33% [P21], Agentless having an issue resolve rate of 32.00% [P13], SpecRover having an issue resolve rate of 31.00%[P19] and AutoCodeRover v2 having an issue resolve rate of 30.67% [P16].

From Table 4 and Table 5, it can be seen that many of the best performers are close-source agents that might not have articles written about their architecture, like Blackbox AI [P14], [P20], Gru [P14], [P20], etc. Still it made sense to add those under this research question since it is talking about what kind of agents exist and what is their performance. Also it can be seen that a lot of the results were extracted from the study during which PatchPilot was developed [P20]

Table 5.LLM agents and their performance (i.e., Issue resolution rate) on SWE-Bench Lite

LLM Agent and LLM (if available)	Issue Resolution Rate
Blackbox AI/Blackbox Ai Agent [P14], [P20]	49.00%
Gru [P14], [P20]	48.67%
Globant Code Fixer [P14], [P20]	48.33%
devlo [P14], [P20]	47.33%
PatchPilot + Claude-3.5-Sonnet [P20]	45.33%
Kodu-v1 + Claude-3.5-Sonnet [P20]	44.67%
CodeStory Aide + Mixed Models [P10]	43.00%
CodeStory Aide + GPT-4o + Claude 3.5 [P13]	43.00%
OpenHands + CodeAct v2.1 + Claude-3.5-Sonnet [P20]	42.00%
Composio SWE-Kit [P20]	41.00%
Agentless + Claude-3.5-Sonnet [P20]	40.67%
Bytedance MarsCode [P20]	39.33%
LingmaAgent + Claude3.5 Sonnet v1022 [P2]	38.33%
Moatless + Claude-3.5-Sonnet [P20]	38.33%
AppMap Navie v2 [P20]	36.00%
Isoform [P20]	35.00%
MarsCode Agent [P19]	34.00%
SuperCoder2.0 [P20]	34.00%
Alibaba Lingma Agent + Claude-3.5-Sonnet [P20]	33.00%
CodeShellTester + GPT-4o [P20]	33.00%
Agentless + GPT-4o [P20]	32.00%
SpecRover + GPT-4o + Claude-3.5-Sonnet [P18], [P20]	31.00%
AutoCodeRover-v2 + GPT-4o [P20]	30.67%
Amazon Q Developer-v2 [P20]	29.67%
RepoGraph + GPT-4o [P20]	29.67%
MASAI [P6]	28.33%
SIMA + GPT-4o [P20]	27.67%
MASAI [P20]	27.33%
Agentless + Claude-3.5 [P3]	26.79%
Agent-101 [P6]	26.67%
IBM Research Agent-101 [P20]	26.67%
Aider [P6], [P18], [P20]	26.33%
HYPERAGENT-Full-1 [P5]	26.00%
OpenDevin [P6], [P18]	26.00%
Lingma & ACR [P2]	25.33%

5.3 Factors influencing the performance of LLM-based agents (RQ3)

There were many factors influencing the performance of LLM-based agents pointed out in the papers, some agents had focused on similar aspects, while others had taken a different direction. The summary table of influencing factors of investigated agents is given in Table 6.

As shown in Table 6, patterns of architectural and design patterns contribute to higher issue resolution rates while having the following factors implemented:

- **Specialized Sub-Agents:** Multi-agent systems excel when compared against single-agent systems, dividing different tasks between components like planners, editors, and validators.
- **Multi-Step Reasoning:** Some of the successful agents plan and attempt issue resolution over multiple steps, revisiting and refining their outputs with implementing searches for decision-making.
- **Repository Understanding:** Agents that use structured code representations have a superior comprehension of code, they improve localizing the bugs and the patch quality.
- **Dynamic Context Retrieval:** Instead of processing the full codebase, only relevant code fragments are extracted using embeddings or graph search techniques.
- **Iterative Feedback Loops:** Some agents validate patches through conducting test runs before submission and increasing its chance of successful resolution of the issue.
- **Multiple Patch Attempts:** Agents allowing retries, achieve marginally better resolve rates compared to agents which do not allow retries.
- **Tool Augmentation:** Integration of external tools that improve performance. Agents that use components like Retrieval-Augmented Generation (RAG) or graph search for navigating large codebases more efficiently and retrieve context that is relevant for fixing the issue.
- **Fine-Tuning and Automation:** Fine-tuned models designed for repair tasks and full-cycle automation pipelines are more successful in end-to-end resolution workflows.
- **End-to-End Automation Focus:** Some of the most effective agents are those designed to effeciently operate autonomously across the entire resolution pipeline on a repository level, starting from understanding the issue to localizing the bug, proposing a fix, testing it, and at the end validating the outcome. Agents using end-to-end automation

demonstrate the benefits of full-cycle automation, reducing the need for human oversight and increasing scalability.

- **Handling Complexity (Modularity):** Dealing with complex or multi-file issues often requires breaking down the problem into smaller, more manageable subtasks. Modular frameworks use multiple interconnected components, like issue reproducers, patch generators, and reviewers, to isolate complexity and improve focus at each stage. This modularity enhances both reliability and success rates.

These insights highlight that agent performance is not solely dependent on the underlying large language models, but also on how intelligently the surrounding agentic infrastructure is designed. The presence of multiple such factors in high-performing systems reinforces their importance in practical applications.

Table 6. Factors improving agents' performance

Factor	Description	Agents benefiting from the factor
Specialized Sub-Agents	Instead of one LLM solving everything, they divide the task between specialized roles like Planner, Searcher, Editor, Validator.	HyperAgent[P5], PatchPilot[P21], MASAI[P6], SpecRover[P19]
Multi-Step Reasoning and Planning	Successful agents plan multiple steps ahead and refine their work iteratively.	HyperAgent (Planner + Navigator + Executor)[P5], LingmaAgent (MCTS search)[P2]
Repository and Codebase Understanding	Deep knowledge of the repo using tools like knowledge graphs or line-level graphs improves patching.	LingmaAgent (Knowledge Graph)[P2], RepoGraph (used in SpecRover)[P12]
Dynamic Context Retrieval	Smart retrieval of only the relevant parts of the codebase using search, graphs, MCTS, or embeddings.	SpecRover[P19], LingmaAgent[P2], RepoGraph[P12], MASAI[P6]

Iterative Feedback/Validation Loops	Running tests, validating patches, and refining solutions multiple times before submitting.	PatchPilot (Refinement Component)[P21], SpecRover (Reviewer Agent)[P19]
Multiple Attempts / Try Again Strategy	Allowing multiple patch attempts greatly boosts success instead of "one-shot" approaches.	LocAgent (up to 10 attempts)[P3], SpecRover (iterative patching)[P19]
Tool-Augmented Agents (RAG/Graph Search)	Integrating code retrieval tools (like RAG, RepoGraph, MCTS) to help with retrieval before generation.	SpecRover[P19], HyperAgent (RAG-enhanced runs)[P5], LingmaAgent[P2]
Fine-Tuning or Specialized Training	Fine-tuning LLMs on repair/localization tasks tailored to code fixing.	LocAgent (Qwen2.5 fine-tuned)[P3], HyperAgent (fine-tuned code agents)[P5]
End-to-End Automation Focus	These systems try to automate the full cycle from understanding the codebase to localizing faulty lines of code to fixing the those lines of code to testing without human input.	HyperAgent[P5], PatchPilot[P21], MASAI[P6]
Handling Complexity (Modularity)	Breaking complex issues into smaller subtasks (modular architecture).	MASAI (Test Template Generator, Issue Reproducer, etc.)[P6], SpecRover (Reproducer + Context Retriever + Patch Generator + Reviewer)[P19]

5.4 Consistency of agents' performance on multiple datasets (RQ4)

In this chapter the focus is on agents that were developed for the primary studies and don't look at the closed-source ones, so the graph would not become cluttered. Most papers did not test agents on multiple datasets. Only on 6 papers out of 25 were the agent that was created for the paper, tested on multiple datasets. Research groups which had created the agent themselves only tested their own agent, with the exception of papers where the focus was on adding tools to already existing agents to see how those influence the performance of the agents. The papers where the agent was created by a

research group that had comparative tables of the experiments already ran on the dataset by other research groups or developers and results confirmed by the creators of the datasets through logs and traces [4]. Because of this, it becomes evident that there is a high likelihood that the experiments weren't ran on the same computer hardware, since none of the papers have mentioned what hardware they used for benchmarking the results. Agents' results can be seen on Figure 4, where each dot symbolizes resolution rate on a dataset.

From looking at the Figure 4 it's clear that the HyperAgent-Full-1 was tested on most datasets, consisting of SWE-Bench Lite, SWE-Bench Verified, Defects4J v1.2, Defects4J v2 and RepoExec [P5]. The HyperAgent was the best performing agent according to their results on each of the datasets [P5]. Since the Defects4J datasets weren't tested on other LLM agents then it can not be evaluated whether the results are as expected. The CodexGraph was tested on CrossCodeEval, SWE-Bench full dataset and EvoCodeBench. They compared their results to AutoCodeRover, BM25 and large language models without RAG [P5]. They tested CodexGraph with Qwen2, DS-Coder and GPT-4o. Out of the benchmarking that they did, the resolve rates with CodexGraph were best out of all of the benchmarking that was conducted.

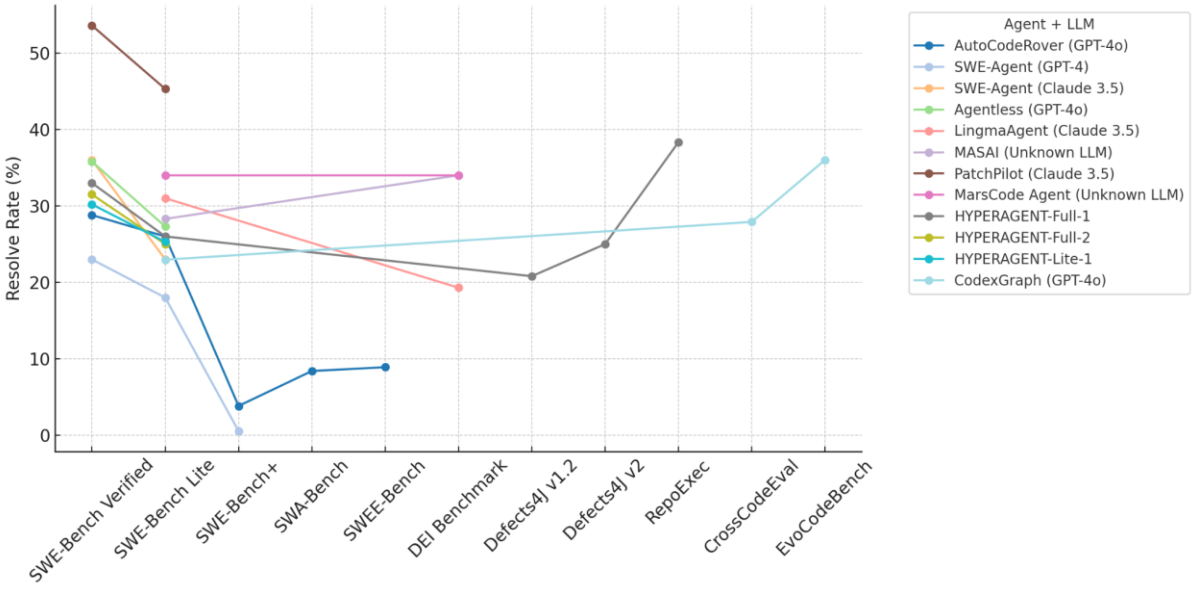


Figure 4. Agents' issue resolution rate across multiple datasets

In the SWE-Bench+ article [P18], they compared combinations of agents and large language on SWE-Bench and SWE-Bench+ and found out that while the best resolve rate for SWE-Bench full dataset was 18.83%, the same combination which was also the

best on SWE-Bench+ ended up getting a resolve rate of 3.83% which is marginally lower. Also the same trend was present in the article of “Automated Benchmark Generation for Repository-Level Coding Tasks” [P26] where they found that SWA-Bench and SWEE-Bench were more difficult benchmarks than SWE-Bench with the only exception of AutoCodeRover getting better resolve rates compared to the SWE-Bench, where SWE-Bench full dataset had worse resolve rates than the others [P18]. Other than that the agents performed similarly across multiple datasets. Also the same trends can be seen on Figure 4.

6. Discussion

This paper conducts a systematic literature review on large language model agents for issue resolution. It answers the research questions about what kind of agents exist, what kind of benchmarks and metrics exist, what are the factors influencing the performance of the agents and how agents perform across multiple datasets. Since this is a relatively new field, with only a limited number of articles available, most of which have been published in recent years. Some of the sources come from companies, while others originate from academic institutions. These entities have different interests and financial resources, which likely influences what they choose to publish.

Due to the absence of a standardized benchmark, most of the available data is not easily comparable. Additionally, almost no one has attempted to run agents developed by others, relying instead on the reported results. A necessary direction for future research into these agents would be to benchmark agents on the same computer hardware, so the results could be properly used to determine their performance. Current findings indicate that the real-world benefits of integrating agents into pipelines are still insufficient. Therefore, at this stage, it would be more reasonable to focus on improving agent performance rather than cost-efficiency. If the dataset used for evaluation is not private, then data leakage can compromise the results.

It is also worth noting that even when the location and cause of a bug are known, traditional language models often struggle to generate appropriate fixes. This raises the question: how does code quality impact a language model's ability to perform bug fixes?

To obtain a more comprehensive understanding of agent performance, benchmark datasets such as SWE-Bench+, SWEE-Bench, and SWA-Bench should be considered for further testing. Definitive conclusions could not be made. Still, it can be said that since those seem to be more complex datasets for large language model agents, getting a good resolve rate on those datasets would most likely mean that the agent performs better in real-life conditions.

Conclusions

The following conclusions can be made from this thesis:

Most commonly used benchmarks include the different datasets from the SWE-Bench benchmarking system. Other benchmarks exist, but their adoption is not widespread. Papers written about newer benchmarks claim that SWE-Bench datasets could have serious issues like data-leakage which could hinder the validity behind the results of benchmarked agents. Other benchmarks are not as popular for researchers benchmarking their LLM agents.

Currently agents that are performing the best on the SWE-Bench Verified dataset are Blackbox AI Agent, CodeStory Midwit Agent with SWE-Search and Anthropic Tools + Claude3.7 Sonnet. On the SWE-Bench Lite dataset best performing agents are Blackbox AI Agent, Gru and Globant Code Fixer. A classification that can be made is whether they are open-sourced or closed-sourced. Closed-source agentic frameworks have demonstrated better issue resolve rates than open-sourced frameworks. In terms of their performance, open-source agents are not far behind, having slightly worse resolution rates.

The performance of LLM-based agents in automating the resolution of issue reports is significantly influenced by architectural and methodological design choices. Multi-agent systems outperform single-agent models based on the papers included in this thesis. Reasoning behind that being modular design of theirs which enables specialization in tasks such as planning, code retrieval, validation, and patch generation. Factors behind their success are multi-step reasoning, dynamic context retrieval, iterative validation loops, and tool-augmented capabilities like knowledge graphs and retrieval-augmented generation (RAG). Also, fine-tuning agents on domain-specific repair tasks and ensuring an end-to-end automation pipeline were found to drastically improve issue resolution rate. These insights underscore the importance of designing agents with domain awareness, modularity, and feedback mechanisms for large language models to have an impact in real-world scenarios.

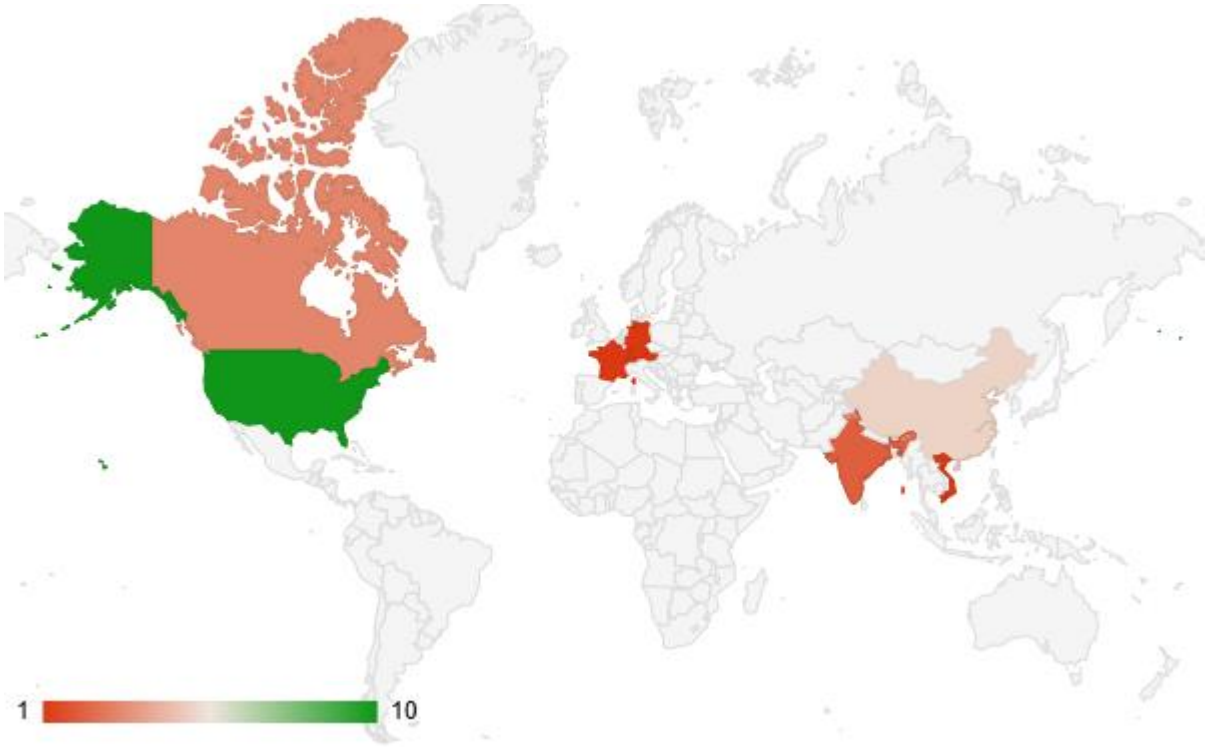
It’s important to acknowledge that only 6 studies out of 25 conducted benchmarking on different datasets and benchmarking systems. The data that is present this far about the consistency of LLM agents’ performance across multiple datasets is mostly consistent, but since the sample size is rather small, it means that the results carry a degree of uncertainty. Most agents performed as expected, with just a few showing slightly unexpected performance. Also, since the results have been verified by logs and traces, the results were

most likely gathered from benchmark runs with different computer hardware, meaning that the developers of the agent ran the benchmark themself, and after that, the results were verified.

Reference List

[1] Z. Zhao *et al.*, “Recommender Systems in the Era of Large Language Models (LLMs),” *IEEE Trans. Knowl. Data Eng.*, vol. 36, no. 11, pp. 6889–6907, Nov. 2024, doi: 10.1109/TKDE.2024.3392335.
[2] X. Meng, Z. Ma, P. Gao, and C. Peng, “An Empirical Study on LLM-based Agents for Automated Bug Fixing,” *arXiv preprint*, arXiv:2411.10213, Nov. 15, 2024, doi: 10.48550/arXiv.2411.10213.
[3] Z. Rasheed *et al.*, “AI-powered Code Review with LLMs: Early Results,” *arXiv preprint*, arXiv:2404.18496, Apr. 29, 2024, doi: 10.48550/arXiv.2404.18496.
[4] C. E. Jimenez *et al.*, “SWE-bench: Can Language Models Resolve Real-World GitHub Issues?”, *arXiv preprint*, arXiv:2310.06770, Nov. 11, 2024, doi: 10.48550/arXiv.2310.06770.
[5] B. Kitchenham, *Procedures for Performing Systematic Reviews*, Keele Univ. Tech. Rep. TR/SE-0401 and NICTA Tech. Rep. 0400011T.1, Jul. 2004.
[6] H. Naveed *et al.*, “A Comprehensive Overview of Large Language Models,” *arXiv preprint*, arXiv:2307.06435, Oct. 2024, doi: 10.48550/arXiv.2307.06435.
[7] C. Liu *et al.*, “Code Execution with Pre-trained Language Models,” *arXiv preprint*, arXiv:2305.05383, May 2023, doi: 10.48550/arXiv.2305.05383.
[8] M. Raza, Z. Jahangir, M. B. Riaz, M. J. Saeed, and M. A. Sattar, “Industrial Applications of Large Language Models,” *Sci. Rep.*, vol. 15, Art. no. 13755, Apr. 2025, doi: 10.1038/s41598-025-98483-1.
[9] F. Zubair, M. Al-Hitmi, and C. Catal, “The Use of Large Language Models for Program Repair,” *Comput. Stand. Interfaces*, vol. 89, Art. no. 103951, Dec. 2024, doi: 10.1016/j.csi.2024.103951.
[10] Q. Feng, X. Ma, J. Sheng, Z. Feng, W. Song, and P. Liang, “Integrating Various Software Artifacts for Better LLM-based Bug Localization and Program Repair,” *arXiv preprint*, arXiv:2412.03905, Dec. 2024, doi: 10.48550/arXiv.2412.03905.
[11] SuperAnnotate, “LLM Agents: The Ultimate Guide 2025,” SuperAnnotate Blog, Mar. 11, 2025. [Online]. Available: https://www.superannotate.com/blog/llm-agents
[12] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models,” arXiv preprint, arXiv:2201.11903, Jan. 2022, doi: 10.48550/arXiv.2201.11903.
[13] 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,” arXiv preprint, arXiv:2305.10601, May 2023, doi: 10.48550/arXiv.2305.10601.
[14] D. Tranfield, D. Denyer, and P. Smart, “Towards a Methodology for Developing Evidence-Informed Management Knowledge by Means of Systematic Review,” *Br. J. Manag.*, vol. 14, no. 3, pp. 207–222, Sep. 2003, doi: 10.1111/1467-8551.00375.
[15] C. Okoli and K. Schabram, “A Guide to Conducting a Systematic Literature Review of Information Systems Research,” *SSRN Electron. J.*, 2010, doi: 10.2139/ssrn.1954824.
[16] A. Thomas, K. Ramesh, and S. Mohan, “Multimodal LLM Agents: Exploring LLM Interactions in Software, Web and Operating Systems,” OpenReview, Apr. 2025. [Online]. Available: https://openreview.net/forum?id=YGLOpASCY5
[17] D. Moher, A. Liberati, J. Tetzlaff, D. G. Altman, and The PRISMA Group, “Preferred Reporting Items for Systematic Reviews and Meta-Analyses: The PRISMA Statement,” *PLoS Med.*, vol. 6, no. 7, p. e1000097, 2009, doi: 10.1371/journal.pmed.1000097.

Appendix I. Countries where contributions have been made towards primary studies



Appendix II. List of Primary Studies from SLR process

[P1] W. Tao, Y. Zhou, Y. Wang, W. Zhang, H. Zhang, and Y. Cheng, "MAGIS: LLM-Based Multi-Agent Framework for GitHub Issue Resolution," Jun. 27, 2024, arXiv: arXiv:2403.17927. doi: 10.48550/arXiv.2403.17927.

[P2] Y. Ma, Q. Yang, R. Cao, B. Li, F. Huang, and Y. Li, "Alibaba LingmaAgent: Improving Automated Issue Resolution via Comprehensive Repository Exploration," Mar. 26, 2025, arXiv: arXiv:2406.01422. doi: 10.48550/arXiv.2406.01422.

[P3] Z. Chen et al., "LocAgent: Graph-Guided LLM Agents for Code Localization," Apr. 29, 2025, arXiv: arXiv:2503.09089. doi: 10.48550/arXiv.2503.09089.

[P4] Y. Xiao, R. Wang, L. Kong, D. Golac, and W. Wang, "CSR-Bench: Benchmarking LLM Agents in Deployment of Computer Science Research Repositories," Feb. 11, 2025, arXiv: arXiv:2502.06111. doi: 10.48550/arXiv.2502.06111.

[P5] H. N. Phan, T. N. Nguyen, P. X. Nguyen, and N. D. Q. Bui, "HyperAgent: Generalist Software Engineering Agents to Solve Coding Tasks at Scale," Nov. 05, 2024, arXiv: arXiv:2409.16299. doi: 10.48550/arXiv.2409.16299.

[P6] D. Arora et al., "MASAI: Modular Architecture for Software-engineering AI Agents," Jun. 17, 2024, arXiv: arXiv:2406.11638. doi: 10.48550/arXiv.2406.11638.

[P7] X. Liu et al., "CodexGraph: Bridging Large Language Models and Code Repositories via Code Graph Databases," Aug. 11, 2024, arXiv: arXiv:2408.03910. doi: 10.48550/arXiv.2408.03910.

[P8] R. Aleithan, H. Xue, M. M. Mohajer, E. Nnorom, G. Uddin, and S. Wang, "SWE-Bench+: Enhanced Coding Benchmark for LLMs," Oct. 10, 2024, arXiv: arXiv:2410.06992. doi: 10.48550/arXiv.2410.06992.

[P9] B. Huang, Y. Yu, J. Huang, X. Zhang, and J. Ma, "DCA-Bench: A Benchmark for Dataset Curation Agents," Jun. 11, 2024, arXiv: arXiv:2406.07275. doi: 10.48550/arXiv.2406.07275.

[P10] A. Gautam, K. Kumar, A. Jha, M. Ns, and I. Bhola, "SuperCoder2.0: Technical Report on Exploring the feasibility of LLMs as Autonomous Programmer".

[P11] A. Antoniades, A. Örwall, K. Zhang, Y. Xie, A. Goyal, and W. Wang, "SWE-Search: Enhancing Software Agents With Monte Carlo Tree Search and Iterative Refinement," arXiv preprint arXiv:2410.20285, Apr. 2025.

[P12] S. Ouyang et al., "RepoGraph: Enhancing AI Software Engineering with Repository-level Code Graph," Mar. 18, 2025, arXiv: arXiv:2410.14684. doi: 10.48550/arXiv.2410.14684.

[P13] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Agentless: Demystifying LLM-based Software Engineering Agents," Oct. 29, 2024, arXiv: arXiv:2407.01489. doi: 10.48550/arXiv.2407.01489.

[P14] Z. Yu et al., "OrcaLoca: An LLM Agent Framework for Software Issue Localization," Feb. 01, 2025, arXiv: arXiv:2502.00350. doi: 10.48550/arXiv.2502.00350.

[P15] Y. Zhu et al., "CVE-Bench: A Benchmark for AI Agents' Ability to Exploit Real-World Web Application Vulnerabilities," Apr. 10, 2025, arXiv: arXiv:2503.17332. doi: 10.48550/arXiv.2503.17332.

[P16] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "AutoCodeRover: Autonomous Program Improvement," Jul. 25, 2024, arXiv: arXiv:2404.05427. doi: 10.48550/arXiv.2404.05427.

[P17] K. Zhang et al., "DIVERSITY EMPOWERS INTELLIGENCE: INTEGRAT- ING EXPERTISE OF SOFTWARE ENGINEERING AGENTS," 2025.

[P18] K. Vergopoulos, M. N. Müller, and M. Vechev, "Automated Benchmark Generation for Repository-Level Coding Tasks," Mar. 10, 2025, arXiv:

arXiv:2503.07701. doi: 10.48550/arXiv.2503.07701.
[P19] H. Ruan, Y. Zhang, and A. Roychoudhury, “SpecRover: Code Intent Extraction via LLMs,” Dec. 11, 2024, arXiv: arXiv:2408.02232. doi: 10.48550/arXiv.2408.02232.
[P20] Y. Liu et al., “MarsCode Agent: AI-native Automated Bug Fixing,” Sep. 04, 2024, arXiv: arXiv:2409.00899. doi: 10.48550/arXiv.2409.00899.
[P21] H. Li, Y. Tang, S. Wang, and W. Guo, “PatchPilot: A Stable and Cost-Efficient Agentic Patching Framework,” Feb. 04, 2025, arXiv: arXiv:2502.02747. doi: 10.48550/arXiv.2502.02747.
[P22] T. Ahmed, M. Hirzel, R. Pan, A. Shinnar, and S. Sinha, “TDD-Bench Verified: Can LLMs Generate Tests for Issues Before They Get Resolved?,” Dec. 03, 2024, arXiv: arXiv:2412.02883. doi: 10.48550/arXiv.2412.02883.
[P23] Y. Ma et al., “Thinking Longer, Not Larger: Enhancing Software Engineering Agents via Scaling Test-Time Compute,” Apr. 08, 2025, arXiv: arXiv:2503.23803. doi: 10.48550/arXiv.2503.23803.
[P24] N. Nashid, I. Bouzenia, M. Pradel, and A. Mesbah, “Issue2Test: Generating Reproducing Test Cases from Issue Reports,” Mar. 20, 2025, arXiv: arXiv:2503.16320. doi: 10.48550/arXiv.2503.16320.
[P25] R. Cheng et al., “Agentic Bug Reproduction for Effective Automated Program Repair at Google,” Mar. 11, 2025, arXiv: arXiv:2502.01821. doi: 10.48550/arXiv.2502.01821.

I. Licence

Non-exclusive license to reproduce thesis and make thesis public

I, Karel Udras,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Automated Resolution of Issue Reports using LLM Agents: A Systematic Literature Review, supervised by Faiz Ali Shah.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Karel Udras

15.05.2025