

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Martin Kilgi 232608IVCM

**EVALUATING THE EFFECTIVENESS OF MULTI-AGENT
LARGE LANGUAGE MODEL SYSTEM FOR AUTOMATED
VULNERABLE CODE REPAIR**

Master's Thesis

Supervisor: Hayretдин Bahsi
Ph.D

Tallinn 2025

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Martin Kilgi 232608IVCM

**SUUREL KEELEMUDELIL PÕHINEVA MITME AGENDI
SÜSTEEMI EFEKTIIVSUSE HINDAMINE HAAVATAVA
KOODI AUTOMAATSES PARANDAMISES**

Magistritöö

Juhendaja: Hayretdin Bahsi
Ph.D

Tallinn 2025

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Martin Kilgi

18.05.2025

Abstract

Until the artificial intelligence isn't perfectly optimized or if code is still written by humans, there will always be vulnerabilities in it. For one vulnerability, there might be multiple ways to fix or mitigate it. Depending on the nature of the security flaw and environment, it may take a fair amount of time to find a suitable solution for it. Even if found vulnerabilities are easy to repair, a developer or engineer still has to commit to it and contribute some time to fully understand the situation. This is a problem because there usually is a high risk of human errors in code and also the work of software engineers or cybersecurity specialists is generally very pricey.

This thesis measures the effectiveness of multi-agent system approach to large language models in repairing real-world software vulnerabilities and providing security-related solutions for them. It aims to provide foundational information on such system in handling code-level security issues. Results of this study are compared to the results of other recent studies focusing on the same topic, using similar data with different approaches like fine-tuning models and prompt engineering. End Results show which approach of using large language models to repair insecure code is more efficient - multi-agent or other analyzed methods. Final analysis also displays whether the state of artificial intelligence is ready to be, to some extent, set responsible for automatic code repair. Possibly, in turn it can reduce the risk of human error in repairing vulnerable patterns and also time consumption that would go into working out efficient solutions. Although there are security tools and scanners from which some offer similar functionality, they are not that scalable and can't keep up with the rapid development of technology as well as AI powered tools.

Considering the results achieved by using artificial intelligence based multi-agent approach for providing repairing solutions for vulnerable code snippets, it justified itself as a valid option to consider for helping with vulnerable code analysis and fixing. Although it does not entirely eliminate the need for human effort, it can provide alternative solutions and ideas for the engineer to work with. With high probability, this will speed up the repairing process and help to see the problem from different angles which in turn will improve the quality of final solution.

The thesis is written in English and is 85 pages long, including 7 chapters, 8 figures and 4 tables.

Annotatsioon

Suurel keelemudelil põhineva mitme agendi süsteemi efektiivsuse hindamine haavatava koodi automaatses parandamises

Kuni tehisintellekt ei ole perfektselt optimeeritud või kuni tarkvara luuakse ikkagi inimeste poolt, ei kao turvanõrkused kuhugi. Ühe turvaaugu parandamiseks või leeven- damiseks on tihti mitu erinevat viisi. Sõltuvalt selle olemusest ja seda ümbritsevast keskkonnast, võib sobiva lahenduse leidmiseks kuluda arvestatav hulk aega. Isegi kui leitud nõrkust on lihtne parandada, peab sellega tegelev insener mingi osa oma aega panustama, et olukorda põhjalikult analüüsida ja seejärel lahendus implementeerida. Kuna inimlikke vigu on võimatu täielikult vältida ning nii tarkvarainseneride kui ka küberturbeekspertide tööaeg on väga kallis, kujutab eelkirjeldatud olukord olulist probleemi.

Selle uurimistöö käigus uuritakse kui efektiivselt suudab suurel keele mudelil põhinev mitmest autonoomsest agendist koosnev süsteem parandada või pakkuda lahendusi päris elust tulenevatele ehk mittekunstlikele tarkvara haavatavustele. Töö üheks eesmärgiks on anda fundamentaalset informatsiooni selle kohta, kuidas taolised süsteemid käsitl- evad koodis väljenduvaid turvaprobleeme. Saadud tulemusi võrreldakse teiste samale teemale keskenduvate ja sarnaseid andmeid pruukivate hiljutiste uuringutega, mis kasu- tavad erinevaid lähenemisi nagu mudelite peenhäälestamine ja mudeli sisendi disainimine. Lõpptulemus kajastab, milline meetod on haavatava koodi automaatselt parandamiseks efektiivsem - kas mitmel agendil põhinev süsteem või mõni muu analüüsitud lähenemistest. Samuti näitab käesolev analüüs, kas tehisintellekt on piisavalt arenenud, et olla valmis võtma vastutust automaatselt koodiparanduste sisseviimise eest. Potentsiaalselt võib see protsess vähendada inimlike eksimuste arvu tarkvaraarenduses ja samuti säästa aega, mis läheks nõrkuseid adresseerivate efektiivsete lahenduste väljatöötamiseks. Ka praegu eksis- teerib erinevaid turvanõrkustega tegelevaid tööriistu ja skännereid, millest mõned pakuvad sarnast funktsionaalsust. Probleem aga on selles, et nad ei ole nii skaleeruvad ning ei suuda kaasas käia tehnoloogia ülikiire arengu ning ka tehisintellektil põhinevate tööriistadega.

Kui käesolevas uuringus kasutatud lähenemine osutub haavatava koodi parandamises efektiivseks, viitab see võimalusele arvestada seda potentsiaalse valikuna, mida kasutada

haavatava koodi analüüsimisel ja parandamisel. Kuigi inimeste tööd see täielikult ei välista, võib see aidata leida alternatiivseid lahendusi ja ideid, millega insener edasi saab töötada. Suure tõenäosusega kiirendab see koodi parandmise protsessi ja aitab näha probleemi erinevate nurkade alt, mis kõik parendavad lõpptulemuse kvaliteeti.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 85 leheküljel, 7 peatükki, 8 joonist, 4 tabelit.

List of Abbreviations and Terms

API	Application Programming Interface
CPU	Central Processing Unit
LLM	Large Language Model
CVE	Common Vulnerabilities and Exposures
SAST	Static Application Security Testing
APR	Automatic Program Repair
CLI	Command Line Interface
JSON	JavaScript Object Notation
URL	Uniform Resource Locator
HTTP	Hyper Text Transfer Protocol
LAN	Local Area Network
SDK	Software Development Kit
DoS	Denial of Service
DDoS	Distributed Denial of Service
SVP	Software Vulnerability Prediction
DNN	Deep Neural Network

Table of Contents

1	Introduction	10
1.1	Research Motivation	10
1.2	Problem Statement	11
1.3	Research Questions	12
1.4	Scope and Goal	12
1.5	Novelty	13
2	Background	14
3	Literature Review	16
3.1	Methods	16
3.2	Usage of Large Language Models in the Process of Writing Code	16
3.3	Multi-Agent Large Language Models	18
3.4	Usage of Artificial Intelligence in Cybersecurity And Software Testing	21
3.5	Usage of Artificial Intelligence in Vulnerability Detection	22
3.6	Existing Work on Artificial Intelligence Based Vulnerability and Bug Fixing Solutions	24
3.7	Summary of Findings	27
4	Research Methods	28
4.1	Experimental Environment	29
4.2	Multi-Agent Framework	32
4.3	Dataset	36
4.3.1	Evaluating Datasets	37
4.3.2	Chosen Dataset	38
4.4	Selection of Large Language Model	40
5	Experimental Flow	43
5.1	Multi-Agent Setup	43
5.2	Filtering the Dataset	47
5.3	Automated Workflow Description	49
5.4	Evaluation Method	51
5.4.1	Unit Tests	52
5.4.2	Manual Evaluation	52
6	Results	54

6.1	Validating Using Tests	56
6.2	Usage of Provided Tools	57
6.3	Results From Previous Studies and Comparison	61
6.4	Ablation Study	64
6.5	Additional Observations	67
6.5.1	Code Formatting and Generating Compilable Files	67
6.5.2	Time Consumption	67
7	Conclusion	69
7.1	Limitations	70
	References	71
	Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis	77
	Appendix 2 – Software Developer Agent and Its Task	78
	Appendix 3 – Security Analyst Agent and Its Task	79
	Appendix 4 – Research Agent and Its Task	80
	Appendix 5 – Code Sanitization Agent and Its Task	81
	Appendix 6 – Code Integrity Agent and Its Task	82
	Appendix 7 – System and User Message of A Single Prompt	83

List of Figures

1	Population growth of ChatGPT compared to other applications measured in days	14
2	Workflow of AI agents processing user input	19
3	Price comparison of DeepSeek R1 and OpenAI o1	41
4	Abstract process of agents' interaction with each other and LLM to resulting final Java file	45
5	Process of manual evaluation of the generated result	53
6	LLM session divided into different states	59
7	Decision indicating web scraping agent tool usage	60
8	Result of web scraping tool targeting vulnerability specific NIST NVD website	60

List of Tables

1	Used Vul4J vulnerabilities grouped by CWE categorizations	48
2	Comparison of results using different configurations with two models . .	55
3	Required modifications to pass Vul4J tests	56
4	Evaluation of different approaches.	65

1. Introduction

1.1 Research Motivation

As the dependence on software in various domains is already high and constantly rising, the quality and reliability of it is becoming more and more important. Putting more responsibility on software also means that there is more to lose during a cyberattack. Average cost of a data breach in 2024 was 4.88 million US dollars which was about 10% more than in previous year. The same report also highlighted that the average cost savings for organizations that used artificial intelligence in their security processes was 2.22 million US dollars. [1] One of the key aspects to avoid successful cyberattacks is to keep the code free of vulnerabilities. Vulnerable code can have large financial, reputational or even lethal consequences. About every 17 minutes, new vulnerability is published and on average it takes 277 days for security teams to identify a data breach. [2] This alone highlights the importance of checking the code for vulnerabilities frequently and ensuring it can't be exploited.

Integrating more software into different domains means larger codebases, higher maintenance and greater chance of containing vulnerabilities. Software developers' and cybersecurity specialists' work is usually expensive and therefore their time should be used as efficiently as possible. There exists different security scanners and code analyzers but most of them are responsible only for vulnerability detection not repairing. A lot of these tools are relying on signatures of different vulnerabilities and are therefore deterministic. This means that only previously acknowledged vulnerabilities and patterns can be detected. Because of that, majority of code repairing process is still manual and time consuming. Study conducted on measuring the performance of APR tools based on real-world Java vulnerabilities [3] concluded that majority of fixing attempts do not succeed. A total of 90.12% out of all repairing attempts failed by which 4.35% was caused by tool failures, 13.62% by timeouts and over 82% due to tools' repairing strategy limitation. [3] These statistics confirm that a large part of current automatic security fixing solutions fail because of their knowledge base that LLM-s have a great possibility to improve. Large language models have lately proven to be quite efficient at code generation related tasks. This knowledge lays a promising base to use LLM-s in the process of repairing vulnerable code. Studies on this topic already exist [4] [5] [6] and their different approaches and usages of models show various results. Therefore, it would be wise to take step further and study how well can multi-agent large language model system fix vulnerable code.

1.2 Problem Statement

A study was done on evaluating effectiveness of large language models in detecting security vulnerabilities [7]. It found that across different LLM-s and datasets, analyzed models managed to achieve an average accuracy of 62.8% and a F1 score of 0.71. Considering these results, used LLM system outperforms other popular static analysis tools like CodeQL. [7] Although there has been improvement in vulnerability detection, the process of automated code repairing is still in an early state because of diversity and complexity of software vulnerabilities. Existing solutions that are based on rule-based system often fail to keep up with complex and new vulnerabilities. Traditional [5] and single-agent [4] large language model solutions' results are variable. A study experimented with using a singular prompt based on predefined template for fixing vulnerable code [5] across multiple LLM-s like Polycoder and J1 Jumbo. It found that from total of 58 500 generated patches, only 3688 repaired their synthetic programs including two CWE-s. Another study researched how effective is a single autonomous agent equipped with different tools [4] in repairing software bugs. It found that out of 854 bugs in Defects4J dataset, their solution managed to fix 164 which slightly outperformed the current state of art method for APR. Although software bugs and vulnerabilities are not exactly the same, they are similar and still comparable. Reasons for such outcomes of experiments with these approaches might be the lack of context and thorough analysis with inputs from different perspectives. This paper aims to address current limitations of automated vulnerability repairing by evaluating the effectiveness of multi-agent LLM system in vulnerability fixing on code-level. Main priority is to determine if the multi-agent large language model system approach to automated vulnerability repairing offers improved efficiency and adaptability compared to current methods.

1.3 Research Questions

To reach the desired end result, the thesis' author will search answers for following questions:

RQ1: How well can large language model multi-agent system generate solutions for vulnerable code?

RQ2: How does the multi-agent approach compare to the traditional approach in the case of automated code repair?

RQ3: How does providing agents tools and additional information about the vulnerability affect the quality of generated results?

RQ4: How does the removal of specific agents influence the quality of generated results?

1.4 Scope and Goal

The goal of this thesis is to evaluate multi-agent large language model system's performance at automated vulnerable code repairing. Experiments will focus on file-level repairing meaning that the system is given the whole source code of file rather than just a vulnerable function. Because of that, the system is also responsible for finding the correct security flawed location in code. Therefore, in this context, the process of automated code repair also somewhat includes analysis and vulnerability locating in the code as multiple agents examine the provided source code, provide different perspectives according to their context and finally generate the concluded outcome. Expected outcome is to determine whether using multiple large language model based agents can enhance the precision and speed of fixing dangerous flaws in code. The result will be compared against other studies using AI based solutions for vulnerable code repair as well as against a baseline result achieved by using a traditional single prompt in this study. Limitations of the study include the availability and usage restrictions of large language models that can be prompted through API and reliance on public datasets. Data wise the scope of the study includes analyzing a public vulnerability dataset consisting of real-world Java vulnerabilities. Key assumptions for this thesis are a high quality, realistic code dataset and access to a sufficiently advanced language model that is capable of code analysis and recognizing vulnerable patterns.

1.5 Novelty

The novelty in this thesis lies in using multi-agent large language model system in the automated process of fixing vulnerable code. Although the usage of artificial intelligence in various domains is novel itself, there already exists studies done on large language models' capabilities in programming [8]. Going into more detail, studies on code repairing with the help of large language models exist [5] [4], but majority of them uses the traditional approach of simply singularly prompting the model or using a single agent. Therefore, research conducted on leveraging different multiple agents, especially in a niche area like vulnerable code repair, is limited. Using numerous independent unique agents, this research experiments with a rather uncommon approach of using artificial intelligence with a purpose of improving automated code repair. Using this technique, the result goes through multiple communication and feedback loops, theoretically continuously improving and therefore being more accurate than the result that is generated by single agent or prompt. It also contributes to the area by providing statistical evidence on the effectiveness of solving code related tasks using multi-agent system. This lays a base for future research in AI driven automated code vulnerability management or other software related problems in general.

2. Background

Artificial intelligence has gained massive attention in recent years. Just a little while ago, a level of which current publicly usable artificial intelligence is currently on, was only usable for small groups of people. Nowadays, popular AI systems are used on a daily basis by huge amount of people from generating ideas about certain topics to usages in different automated systems. According to Exploding Topics, ChatGPT passed the mark of a million users in just five days after its launch and as of the beginning of March it had over 180 million active users. Figure 1 displays the time taken in days to reach a million users of ChatGPT along with other popular applications for comparison. [9]

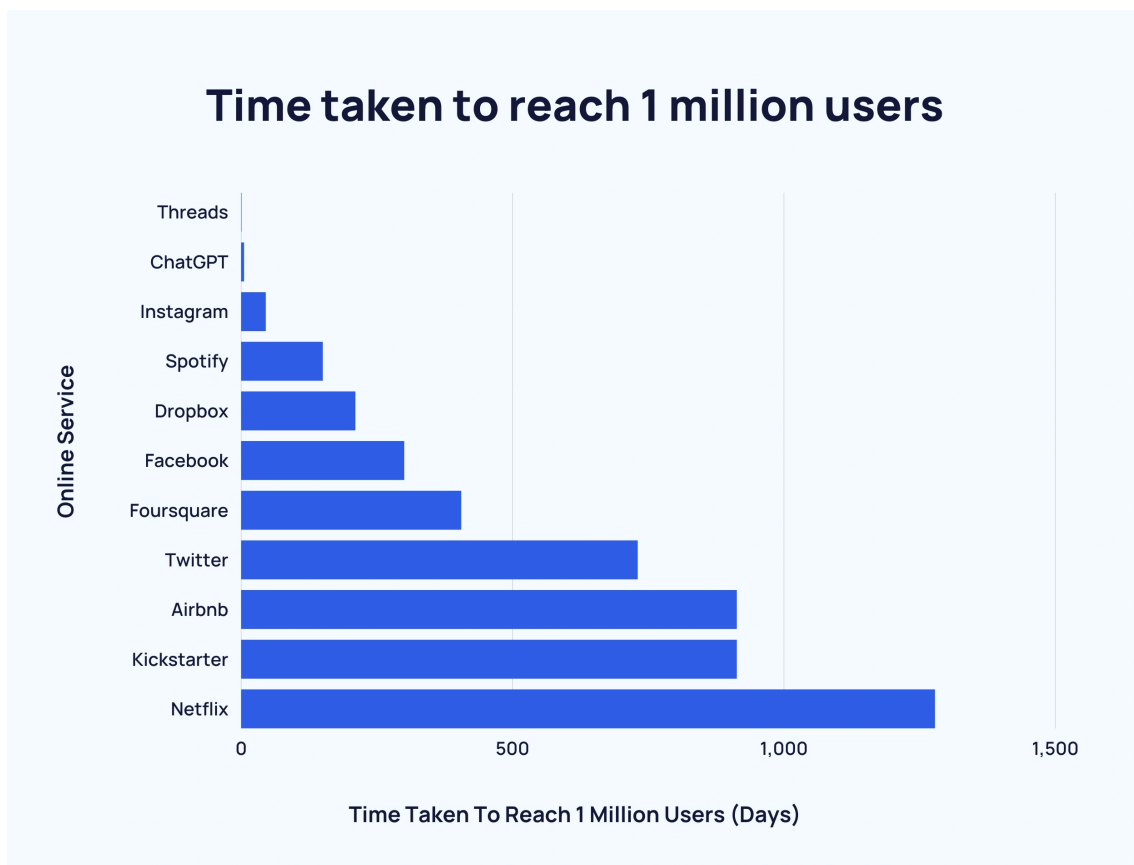


Figure 1. Population growth of ChatGPT compared to other applications measured in days

Many different fields and industries benefit from using large language models in some way, usually by reducing proportion of human participation of the activity. Good candidates for this are from the sector of information technology. Usually, the salary of people who work as a software developer or cybersecurity specialist is well above average and therefore their

time at work is much more important. AI can do things like help developers write code, detect intrusions and test software which results in companies cutting costs. According to an article, software testing takes up 15% - 25% of total project budget [10]. It means that testing can almost make one fourth of the whole estimate. If AI could handle even a half portion of it, it would already be a big win for a company.

What does the term "artificial intelligence" actually mean? According to the study on the state of art of artificial intelligence, it can be defined as a branch of study which consists of many different disciplines that simulates and learns intelligent human behavior using computers. In simple terms, AI works by accepting some amount of data, analyzing and studying the provided data and then storing the acquired knowledge. After that it tries to generate the most formal and correct response it can based on previously learned information. [11]

Its history goes back to 1940s where the first artificial neuron model was presented and 1956 where the term "Artificial Intelligence" was first proposed. Next remarkable period was 1970s when the concept of knowledge engineering was introduced. It helped to proceed with developments, but also introduced new problems like obtaining knowledge from expert systems. That reduced the speed of studying this area gradually until 1982 when the backpropagation algorithm and Hopfield neural network was introduced. This created a new spark and rapidly increased the development the branches of artificial intelligence like speech translation and recognition. Unfortunately, this was also the limit of the development of this time. Then the study of AI was again more in the background until 2006. At this time, better hardware started to get more available and affordable. Also, the storage capacity got much larger which is very important in training and using artificial intelligence. From that time until now, AI development has been very rapid and is constantly being bettered. [11]

Today, technologies based on artificial intelligence are evolving constantly with major breakthroughs being announced almost weekly from topics like generating images to self-driving cars. The rapid development of artificial intelligence has enabled it to be utilized in various areas, including software engineering, for automating tasks as well as filling an assistant's position. When guided and prepared correctly, AI systems are already able to perform analysis on source code, generate properly structured code snippets and detect vulnerabilities. Therefore, it is important to keep up with this technology's development by constantly researching and studying its possibilities using novel approaches.

3. Literature Review

3.1 Methods

This literature review follows the traditional approach of literature review. To search for topical studies and articles, terms like 'artificial intelligence', 'large language models software development', 'multi-agent LLM', 'artificial intelligence cybersecurity', 'software vulnerability detection', 'static application security testing' were used. Google Scholar was used as a main search engine which resulted in finding sources from databases like ACM Digital Library, IEEE Xplore and ScienceDirect. Since the area of artificial intelligence has improved massively in recent years, it was ensured that selected articles were not published before 2015. Many articles providing accurate numerical data were found by studying references of other papers using the snowball method. Following topics were predefined to direct the research: usage of large language models in the process of writing code, multi-agent large language models, usage of artificial intelligence in cybersecurity and software testing, usage of artificial intelligence in vulnerability detection and existing work on artificial intelligence based vulnerability and bug fixing solutions.

3.2 Usage of Large Language Models in the Process of Writing Code

Majority of bigger domains like healthcare and logistics have significant parts that are entrusted to the software. Therefore, there is a specific software that needs to be developed and maintained. If earlier humans thought how to delegate their tasks to software then now we are taking the next step and actively seeking how to delegate developing software to computer. This domain is actively being explored and developed in hope to make developing software more automated and use less human resources. Already at this point of time, large language models themselves can generate code snippets on specific prompts. Quality and reliability of the code heavily depends on the technique and model used, but even the freely accessible smaller models can handle easier coding tasks pretty well.

A study conducted in 2022 compared code generation performance of different models which also included large code-focused language models like Codex and CodeParrot [12]. During the study, they also built their own model named PolyCoder that uses GPT-2 as its model architecture and is trained on 12 different programming languages. Surprisingly, PolyCoder outperformed every other compared models in language C. Compared to similar

size model GPT-Neo, both having about 2.7 billion parameters, PolyCode outperformed it also in languages like Rust, TypeScript and Javascript. One of the ways they compared models was benchmarking them on HumanEval dataset. HumanEval Benchmark dataset is specifically designed to test the competence of code generation of different large language models. It includes over 150 prompts with descriptions of desired code, variable names and function definitions with arguments. Accordingly, it contains test cases to later check if the generated result is valid according to the provided information in the prompt. [13] In general, Codex still gave the best results on all programming languages except C. [12]

The quality of an LLM's output is thought to depend on the model's size and training time. However, a smaller Codex model (300 million parameters) performed well on the HumanEval benchmark, surpassing other models. This shows that in specific domains, the size of a language model might not be as important. The study found that, across multiple programming languages, the model trained on both natural language text and code performed slightly better. This was compared to the model trained exclusively on code. This indicates that for a model with broader reach in programming languages, it might be more reasonable to train it with both natural language and code data. [12]

One area in software development where artificial intelligence manifests is existing AI based code completion tools like Github Copilot, OpenAI Codex and Tabnine. Code completion tools are usually integrated into integrated development environments. They try to understand current context and take into account programming language's syntax and patterns to then offer solutions and snippets to complete currently writeable code. [14] A study evaluated the ability of code repairing of Codex large language models. They chose CWE-787 and CWE-89 as their base synthetic examples. They specified the beginning of the vulnerable programs relevant to these CWEs and inserted it into large language model which resulted in many different and unique vulnerable programs. After that they evaluated the security level and functionality of code with code analysis engine called CodeQL and unit tests. Finally, they extracted programs that were functional but still vulnerable to test the code repairing capabilities of LLMs. [5]

For sample programs generation they used two OpenAI Codex engines which resulted in generating 22 unique vulnerable programs containing CWE-89 and 95 vulnerable programs containing CWE-787. As a result, 491 of a total 22034 suggested solutions fixed the vulnerability of CWE-787 and 3197 of a total 10796 generated patches fixed the vulnerability of CWE-89. It shows that large language models indeed can repair vulnerable code. However, the resulting numbers are not anything too impressive. During the experiment, it was also noted that LLMs can generate non-vulnerable code relatively effectively if they understand the prompted task well. The authors of this paper concluded

that, despite previous positive results, the state of the art in code repair using large language models is limited. Their opinion is that these models are not yet ready to deliver real value in a program repair framework. [5]

Another study was done on how well can artificial intelligence generate competition-level code [15]. Competition-level code references to a code written in competitive programming which usually solves a specific given problem, uses different algorithms and has to be as efficient as possible. They introduced a code generation system called AlphaCode. Models that it used had 41 billion and 9 billion parameters and were trained on 715 gigabytes of human code from GitHub. They were later fine-tuned on 2.6 gigabyte dataset of competitive programming problems called CodeContests. To evaluate the performance of AlphaCode, they compared it against programming competitions available on Codeforces. As a result, AlphaCode got an average ranking in the top 54.3% against other real humans and solved 66% of given problems with first try. This means that it performed better than half of all competitors. The whole test was conducted so models didn't take the core logic from the training data, but came up with its own new solutions for problems. [15]

It represents that large language models can understand hard and complex code. This hugely benefits the capability of repairing vulnerable code because it also requires the model to understand the context and complicated code relations.

3.3 Multi-Agent Large Language Models

If most people are familiar with just prompting large language models through web, there are multiple other ways to use those models. One of the approaches is to use a LLM based agent. Agent in this context is an autonomous entity that can learn, decide and act independently to fulfill its task and goals. They are often each specified with distinct capabilities and objectives which makes them stronger in specific context. To advance this approach, multiple previously described agents with different characteristics can be created and combined into one unified system that is called a multi-agent system. Multi-agent references that two or more agents communicate and work together to reach one common goal [16]. For example, let's say that the user needs to write an example of analytical document. The first agent could be the writer who focuses on the content and main points of the text. Second agent could be the designer who chooses the fonts and builds the template where the content fits in. Lastly, third agent could be the final reviewer who looks at the document with a critical look, tries to find flaws and provide solutions for them. Figure 2 [17] abstractly describes how a team of artificial intelligence based agents handle user requests.

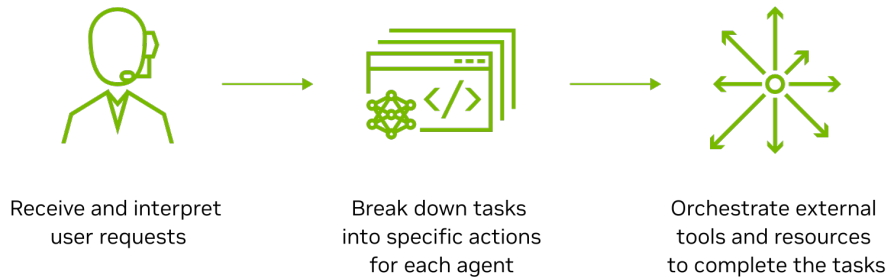


Figure 2. Workflow of AI agents processing user input

Although the first impression of multi-agent system is very promising, it can't be instantly confirmed that it's the best approach for every scenario. A study conducted experiments on both multi-agent and single-agent large language models to measure their reasoning abilities [18]. For this they carried out multiple tests in the form of discussions. Discussion in this case refers to an interactive process where a LLM analyzes and processes a question. In the case of multi-agent system, these agents bounce their opinions back and forth, thus progressively improving the final answer for the given task. For the initial understanding, the study used FOLIO-wiki dataset to analyze both methods [19]. It was found that discussion with a single LLM with large number of parameters combined with a strong prompt produces similar result as the multi-agent LLM discussion. Some of the tasks that were given to large language models were preceded with detailed descriptions of the task. In the case of tasks that didn't include demonstration, multi-agent approach to LLMs outperformed single-agent method. It was also discovered that in multi-agent discussions, performance of agents using weaker language models was progressively improved by agents using more powerful LLMs. [18]

Another paper reviewed the development of large language model based multi-agents over multiple domains that can be categorized into two main categories - problem solving and world simulation [20]. Problem solving included software development and science experiments, world simulation included gaming, economy and society. One of the highlighted

challenges in this paper with LLM-based systems is hallucination. It refers to the fact that one agent generates nonfactual information that the next agent accepts and expands on it. Although this is probably a bigger problem in simulation processes, it still has to be addressed also in code generation related tasks. Second main challenge the study brings out is the evaluation and benchmarking of LLM multi-agent systems. It states that current research focuses on assessing individual agents reasoning and understanding in narrow scenarios. This is the area the author hopes to contribute to with his thesis. Although this paper doesn't conduct any specific experiments on LLMs, it does list quality datasets across different domains, multi-agent implementation tools and other resources. This knowledge is very useful when planning how to setup multi-agent LLM environments and conduct experiments. [20]

3.4 Usage of Artificial Intelligence in Cybersecurity And Software Testing

With artificial intelligence being so widely used in software development, it's not a surprise that it is also popular in the field of cybersecurity. As parts of those two domains overlap, the application of large language models usage also overlaps in some areas. Cybersecurity is a field that has to always be aware of most recent news, must keep up with the time and be able to scale well. Because of that, it sets a very promising ground for maximizing the potential of artificial intelligence.

So far artificial intelligence in cybersecurity has had a great success in different intrusion detection systems. Intrusion detection system is a tool for monitoring and inspecting network traffic to find anomalies and possible malicious acts [21]. Advantage of using AI in those systems comes from its ability to intelligently examine passing traffic and group it into sections. First and maybe with most easily understandable impact attack method is denial of service attack, often called DoS attack. It usually works by commanding a lot of devices to repeatedly make requests to specific machine. It will be proceeded until the targeted machine becomes overloaded, resulting in crashing and becoming unavailable. A study conducted in 2016 applied a deep learning based Distributed Denial of Service attack detection system. It was implemented in Software-Defined Networking environment. It was noted that the mentioned system could detect DDoS attacks with an accuracy of almost 96% while having low amount of false-positives compared to other similar works. [22]

As AI can digest a lot of data and make conclusions out of it very quickly, it is also pretty competent in more non-technical areas like detecting fake information. A paper published in 2015 characterized a dataset consisting of 4.4 million public posts generated on Facebook. It was found that it included over 11 thousand malicious units that had URLs in them. Authors of this paper proposed their own dataset which was based on post's metadata, entity profile, content and URL features to detect malicious Facebook posts in real time. Later this very dataset was used to train multiple machine learning datasets which reached an accuracy of almost 87% of detecting potentially harmful public posts. [23] If we were to use the method of machine learning to find vulnerable patterns in code, the result of previously mentioned study indicates that it could be quite successful.

In the context of detecting and repairing vulnerable code with the help of AI, static application security testing tools provide very similar functionality. Static application security testing (SAST) tools analyze codebase of the application without executing it while trying to detect vulnerabilities and flaws. Since SAST tools apply fixed vulnerability

detecting techniques, it often results in high number of false positives. They also have a weakness of finding configuration errors and are difficult to use for various types of security vulnerabilities. [24] Study conducted in the end of 2021 compared the performance of static application security testing tools and using the approach of software vulnerability prediction (SVP) based on LLM. Results show that using the method of software vulnerability prediction scores better in both finding and assessing vulnerabilities than the standard SAST tools. For example, analyzing code from dataset called Juliet, SAST tool Flawfinder found security flaws with a precision of 0.507. At the same time, software metrics features using SVP model found vulnerabilities with a precision of 0.843. [25] To some extent, this indicates that classical SAST tools can't keep up with new technology and therefore AI-based solutions should be studied more deeply.

3.5 Usage of Artificial Intelligence in Vulnerability Detection

Artificial intelligence and machine learning have also made a big impact in detecting vulnerabilities in software. Traditional methods and tools for automatic security flaw detection are mostly based on identifying specific vulnerability signatures and performing static code analysis. Therefore, they often fall short in identifying new and obfuscated vulnerabilities.

A study conducted in 2020 [26] searched an answer to a question of how well can state-of-the-art techniques that are based on deep learning perform on detecting real-world vulnerabilities. It found that applying real-world scenarios, these techniques' performance drops more than 50% compared to other previous studies that achieved an accuracy as high as 95% in vulnerability detection. For example, a technique called VulDeePecker [27] achieved an F1 score of 85.4% in its origin study while when applied to real-world scenarios, it resulted in a much lower of 12.18% F1 score. Respectively, it reached a precision percentage of 86.9% previously and 11.12% in the currently handled study. These results originated from using pre-trained models which led to average of 73% drop of F1 score. However, even if the models were retrained with real-world data, F1 score's difference was still about 54%. Main reasons for these results were identified to be related to training data and model selection. Analyzing these reasons revealed that the approaches frequently did not learn the characteristics of vulnerabilities but rather unrelated information like variable names. To improve the state of deep learning based vulnerability detection, the study proposes its own framework called ReVeal that addresses previously found problems in model design. Using the ReVeal pipeline, compared to analyzed state-of-the-art approaches, the precision percentage improved by 33.57% and recall percentage increased as much as 128.38%. Although this study uses more simplistic

deep neural network models than the currently available large language models, it still shows great potential and constant improvement for using AI in vulnerability detection. [26]

Another study was done on vulnerability prediction [28], but instead of using deep neural network models, it used currently popular large language models GPT-3.5 and GPT-4. Due to much higher cost of using GPT-4, the paper primarily focuses on GPT-3.5 while conducting additional experiments with GPT-4. Multiple experiments were done with different configurations like providing the task description to LLM in one run and providing vulnerable code examples in another run. Best result came from providing large language model the task description along with similar code examples from training data to aid it in its decision process. With this configuration, GPT-3.5 managed to achieve the highest F1 score of 54% and recall score of 47.2%. Using a similar approach, but with combining top 3 similar code examples with another three of randomly selected samples, it managed to achieve a F0.5 score of 62.8% and an accuracy of 62.7%. Compared to CodeBERT, another pre-trained programming focused LLM, GPT-3.5 outperformed it in terms of accuracy, precision and F0.5, but underperformed in recall and F1 scores. Therefore, results of these two models were similar. Limited experimentation done with GPT-4 revealed that given the task description and code examples from CWE types, it resulted in a F1 score of 76.4%, F0.5 score of 74.8%, recall score of 79.3% and an accuracy score of 75.5%. While outperforming CodeBERT by 34.8% in accuracy, it also outperformed GPT-3.5 in previously mentioned categories with quite a big margin. This indicates the importance of choosing the base model. Overall, the results give a rather promising impression on using large language models in software vulnerability detection. [28]

In summary, the analyzed studies present both the limitations and also a promising potential in using AI in security flaw detection process. Pure deep neural network models didn't perform very well on real-world vulnerabilities, but using LLM-s provided much better results. As seen in both scenarios using DNN models or LLM-s, either experimenting with different model training methods or with various input designing approaches, they can significantly change the end result. Based on these findings, utilizing artificial intelligence, especially large language models, can have a great impact on automating vulnerability detection and hopefully on other similar fields. Therefore, it should be studied further and experimented with using different approaches.

3.6 Existing Work on Artificial Intelligence Based Vulnerability and Bug Fixing Solutions

As said before, artificial intelligence has already become a big and promising part of software engineering and cybersecurity. Human mistakes in software engineering are inevitable. These errors are easy to make as misplacing few characters or simply forgetting to set a certain attribute can already cause a vulnerability. In software, regardless of the error's simplicity, it can do a lot of damage whether it's financial, reputational or other. Since it is already studied that large language models can understand and write code, it would be wise to apply them also in vulnerability and bug fixing processes. Therefore, the following studies were analyzed where artificial intelligence is used in both repairing vulnerable code as well as in fixing bugs in code.

When this thesis is focusing on the multi-agent large language model approach to repair code, a study conducted engineered a single autonomous agent to do it [4]. Autonomous agent can plan and execute actions while using different external tools to fix a bug. This is somewhat similar to the action of multi-agent LLM because it includes multiple steps and states to finally solve a task. They concluded that their built autonomous agent named RepairAgent outperformed previously registered as a state of art of automated program repair process called ChatRepair. For data the study utilized a set of reproducible software bugs called Defects4J. Out of the 835 bugs present in the dataset, RepairAgent was able to fix 164 of them while ChatRepair fixed 162. This makes the RepairAgent's successful repair percentage 19.64% and ChatRepair's 19.40%. Additionally, RepairAgent's result included 39 bugs that could not be fixed with prior techniques. Although this confirms the capability of LLM-s in code repairing tasks, the process of evaluating agent's performance is currently more valuable. Repairable code snippets have test cases to check if code is problem-free or not. Generated solutions are categorized into two categories - plausible and correct solutions. If the fix passes test cases, it is considered plausible. If it also syntactically matches the fix created by a developer, it is considered correct. This evaluation technique will be considered when building the evaluation process for this thesis. [4]

To measure the effectiveness of available pre-trained models code vulnerability fixing, another study was conducted in 2023 [29]. The aim of the study was to bring out advantages and disadvantages on performing automated vulnerability repair with multiple different pre-trained large language models. Criteria for selecting models were that it must be publicly accessible and trained on fairly large programming language corpus. Chosen models were: CodeGPT, CodeBERT, GraphCodeBERT, UnixCoder and CodeT5. For datasets they used two vulnerability datasets - CVEFixes and Big-Vul. Both of these datasets actively gather

information about vulnerabilities by crawling public CVE databases and open-source projects. To accurately evaluate selected models, a transformer neural network model for repairing software vulnerabilities automatically called VRepair is taken as base. As an evaluation method, this paper referred to prior studies that used previously discussed measuring plausible and correct patches technique. It stated that this method should be preferred, but it requires matching test cases to use it. Unfortunately, datasets used in this study, CVEFixes and Big-Vul, don't have test cases and therefore this evaluation technique couldn't be used. Instead, it uses percentage of perfect prediction accuracy to measure the performance of vulnerability repairing. [29]

As results, chosen pre-trained models were over 16% more accurate than VRepair and had an average prediction accuracy of almost 39% over all five models. Experimental results of models independently were following: CodeGPT 37.93%, UniXcoder 40.62%, GraphCodeBERT 37.98%, CodeBERT 32.94% and CodeT5 44.96%. One belief why VRepair underperforms pre-trained models is that VRepair uses word-level tokenization while other models use subword-level tokenization. Another reason could be that chosen models are trained on much larger codebases than VRepair. The study also included fine-tuning three of the models (CodeT5, CodeBERT and CodeGPT) and comparing the outcome with traditional approach. Improvement varies in the range of 12% to 33%. Being more specific, models' accuracies were improved as follows: BERT architecture 22.51%, T5 architecture 12.02% and GPT architecture 31.33%. [29]

Next, another three different studies using real-world Java vulnerabilities as data were analyzed. The first study [6] uses OpenAI's model GPT-4 [30] that is their most advanced model to assess its capabilities in fixing real-world software vulnerabilities using Vul4J [31] dataset. The main research question of this study is how well can GPT-4 automatically repair real-world vulnerabilities in practical applications. Its evaluation process includes conducting manual review along with using tests provided by Vul4J. The study uses the method-level approach meaning that the vulnerable function is extracted from the file and passed to the LLM. Additionally, the Vul4J dataset [31] was filtered to only contain vulnerabilities that exist in the scope of a single file. This means that all the vulnerabilities in the dataset that are formed across two or more files were excluded. As a result, GPT-4 [30] achieved a promising average fix rate of 33.33%. [6]

Second study [32] done on the automatic vulnerability repair using Java vulnerabilities proposed their own neural network model named VulMaster that thrives at generating vulnerability fixes using data-centric innovation. VulMaster uses a combination of two large language models for its work - a fine-tuned CodeT5 as its base model and ChatGPT as an additional LLM for helping with relevant inputs to the main model. Authors of the

paper believe that this approach utilizes both models' strengths as the fine-tuned CodeT5 can be thoroughly tailored for specific needs and ChatGPT has strong capabilities in generating code when given a well-defined context. Their research uses large C/C++ dataset with over 5800 functions as its main data source, but also utilizes Vul4J [31] in an additional evaluation results process. The paper's evaluation process consists of generated code manual verification and using test cases. This study also uses only single-hunk vulnerabilities, filtering out every other instance that does not meet this criterion. The study's own provided model, VulMaster, managed to achieve a successful repair rate of 25.7%. [32]

Lastly, another study [33] was done on using code focused large language models for automatic vulnerability repairing. This study's code repair process is based on fine-grain cue optimization and local sensitive fine-tuning of the models. With optimizing the cue in detail, the research aims to solve the problem of LLM-s often not receiving enough contextual information about the source code for repairing it. The provided method integrates hints like vulnerability type and repair method into large language model input in a similar manner to this paper. Similarly to the last two analyzed studies, this paper also uses Vul4J [31] dataset as one of its data sources. As an evaluation method, the study uses only manual verification by calculating the accuracy of the generated patch. The best result was achieved by using local sensitive fine-tuning method on CodeT5 model combined with multiple hints which managed to generate correct patches for 10 Vul4J vulnerabilities. [33]

Previously summarized results present a fairly promising statistics for using different language models for code repairing. They also include processes of how the performance of large language model code fixing is evaluated and which datasets are used in automatic code repairing studies. This contributes to both choosing the evaluation method and dataset for this thesis. Due to different approaches and models, outcomes of previous studies vary, indicating the need for more experimentation. Overall, the results are still promising and the impact of applying artificial intelligence to different areas in software development and managing, as well as it repairing, is fairly large. Therefore, this domain appears to significantly benefit from further studying and improvement.

3.7 Summary of Findings

This literature review highlighted progress, potential and experimental results of large language models in technological domains like code generation, code repairing and programming. Studies conducted on using large language models for generating code showcased their good performance and potential for future research. Existing work on current AI based vulnerability fixing solutions also provided fair results. While leveraging the traditional approach of using LLM-s didn't show too good results then using an autonomous agent with additional tools outperformed current state-of-art tool. This motivates to discover further and test for example multi-agent system approach in this field. At this point in time and up to author's knowledge, there are no public papers applying multi-agent approach to vulnerable code repair process. Exploring multi-agent large language model systems in different domains presented mixed results. Using model with large number of parameters, well-constructed prompts and detailed context, single agent system provided similar quality results as multi-agent system. In more of an unknown environment, multiple agents still outperformed standalone LLM. While multiple agents aren't always necessary in tasks based on artificial intelligence, current research shows significant potential for automated vulnerable code repair to benefit from their use.

4. Research Methods

This thesis' research is based on the quantitative experimental research method that allows to evaluate generated results with manual reviewing and tests. The main process of research involves developing the multi-agent large language model system tailored for fixing software vulnerabilities and evaluating its effectiveness and capabilities in the mentioned process. The whole system is based on a singular large language model that is interacted with through API. Setting up, configuring and running the main system is done by using Python and a multi-agent platform called CrewAI [34]. CrewAI provides an open-source framework for orchestrating multiple AI agents that simplifies the process of assigning roles, context and goals for different agents. It also helps with the interaction between the agents, running the whole application and monitoring it in real-time. [34]

For the required data which in the context of this work is vulnerable code snippets, the author is utilizing a publicly available dataset. There are several datasets available like DiverseVul [35] and Big-Vul [36], but this thesis uses a dataset called Vul4J [31]. Vul4J dataset is constructed especially for the purpose of studying the effectiveness and capabilities of automated program repairing. It consists of reproducible real-world Java vulnerabilities which each have specific test cases proving the existence of vulnerability in the file. [31]

On conducting the process of evaluating the validity and correctness of generated results, author primarily uses manual verification along with tools and data from Vul4J dataset [31]. Since this dataset has proof-of-vulnerability test cases that can be used to check if the vulnerable part of the code still exists or not, they are used as one of the methods for evaluation. For each vulnerability, before test cases get executed, it is checked if the generated file fits into the vulnerable project and if the project compiles with it. If the compilation is successful, only then will the tests be executed. Additionally, if the result provided by the system does not compile with the existing project or do not pass tests, it gets checked manually. It is then be compared to the official human patch and evaluated whether the system progressed in right direction or not. Consequently, results are divided between four different statuses - complete, highly relevant, minimally relevant or failed. Experiment is run multiple times with different configurations and differently sized large language models. For example, one run is conducted without giving agents any hints or additional information and during another run agents have access to different tools and are provided information about the vulnerability. For retrieving the base to compare the end

results to, an iteration of simply prompting a LLM without agents is conducted. In the end, the results achieved from not using multi-agent setup and also results from other similar studies are used to compare the main results of this paper from using multi-agent system.

4.1 Experimental Environment

To start experimenting with multiple large language model powered agents, a compatible environment is required. Although it is possible to build a whole system that handles the processes of registering agents and orchestrating their communication, there already exist frameworks that take care of it. Developing it from scratch would take a lot of additional time and research, therefore leaving less time for focusing on the paper's main topic. Because of that, the author decided to use pre-built multi-agent platform called CrewAI [34] whose choice is justified in the multi-agent framework selection section of the paper. With the help of CrewAI framework, it is possible to create custom agents with different roles, goals and a specific background. [34] Using a multi-agent framework simplifies the overall management of the agents that includes their communication and system orchestration.

Some of the more developed large language models can be used through their own UI or website for free, but using them through the API is usually priced. The initial setup and development of the system requires rather large amount of interaction with the LLM. Therefore, if large and developed models are not available, it is wise to sacrifice model's capabilities and choose LLM that is usable for cheap price or for free. The author of this paper decided to go with setting up a local LLM on his own computer. This provided the freedom to configure the connection to the model and ensure that the whole environment is adjustable and controlled. The model used was Meta Llama 3 with 8 billion parameters provided by Ollama [37]. The setup process of the LLM was rather straightforward and simple - install Ollama with downloading the installer from their website, configure preferred port and URL and run `ollama run llama3` in terminal. With that, the model was running on the specified port and could be prompted right away on URL `"/api/generate"`. [38]

```
curl -X POST http://localhost:11434/api/generate -d '{
  "model": "llama3",
  "prompt": "Why is the sky blue?"
}'
```

Although the model has only 8 billion parameters, it still requires resources to run it. Therefore, the model was running on PC equipped with Intel I5-12600K processor and GTX 1070 graphics card while the multi-agent system was developed on independent laptop. Since they were both in the same network, the laptop could prompt the LLM running on PC without any restrictions. Running the model locally gave the user the opportunity to build the initial working application that could then later be improved during experimentation. This approach saved the author from worrying about the costs of using large language models or waiting behind other users in queue that would have been real problems when models from other providers would have been used.

To possibly enhance the performance of multi-agent system automatically repairing vulnerable files, author created a custom vulnerability analysis tool. The tool is based on static application security testing (SAST) tool called Bearer [39] that is built to scan and analyze source code to find and prioritize security risks. Bearer has two options - a free command line based tool called Bearer CLI that is going to be used in this paper and a commercial solution Bearer Pro. Bearer CLI supports languages like Python, Javascript, Go and Java. Since the dataset used in this thesis consists only of Java vulnerabilities, this tool suits the system well. It scans the provided code for vulnerabilities and security risks that cover both OWASP Top 10 [40] and CWE Top 25 [41] lists. These lists include vulnerability categories like access control, injection, security misconfiguration and cryptographic failures. It offers multiple configuring options to adjust the tool to your needs. For example, this includes switching between scanner types, ignoring specific findings, limiting scanning rules and changing the output format. [42] After installing it following the instructions in its Github repository, it can be executed simply by inserting the following command to command line interface (CLI): *bearer scan <filename>*. After the tool finishes its analysis, it outputs a report containing rules that the code was run against, every detected finding with a file and lines location that triggered it and a statistical summary of the findings. The custom tool takes vulnerable file path as an input, creates the conditions and executes Bearer CLI tool with provided path in command line and returns the outputted analysis report. This will then get parsed to the specified agent to help it analyze vulnerable code in process. [39]

As mentioned in the section above, dataset called Vul4J [31] is being used in the study for data with its framework and tools. Its platform requires multiple dependencies like certain versions of Java, Maven and Python. Additionally, it includes built-in SAST tool that can be executed with framework commands that is also an extra dependency. Therefore, rather than setting it all up on a local computer, it was easier to use their official Docker [43] image [44]. Docker is a free-to-use platform that enables developers to build, package, and run applications in isolated environments called containers. Containers, in turn, are units

of software that pack the required code with all its dependencies, creating an independent environment for the application to run in [45]. Taking a simple Java application for example, the container would include a runtime environment such as lightweight Linux operating system or Java runtime image, the actual application code, its dependencies and different configuration files. As all mandatory components are packaged into one object, this eliminates the need for the user to set up those things himself and avoids potential problems originating from different environment settings. That means that it is possible to run containerized applications anywhere with Docker installed independently of the host operating system and existing dependencies and configurations. Image in the context of Docker is a description of what should be put into and executed in a container. In a simplified way, Docker container is just a single running instance of Docker image. [43] As mentioned before, Vul4J fortunately has a pre-built Docker image that already has required modules installed and was almost out-of-the-box solution with minor modifications. [31]

To analyze agent interactions and outputs in more detail, a tool called AgentOps [46] is used. AgentOps is an observability and development tool platform for AI agents designed to help developers build, monitor and assess AI agents. It offers a Python SDK to implement it into a Python program. It integrates with a lot of popular multi-agent frameworks and offers features like debugging by displaying graphs about step-by-step agent execution and managing costs by tracking spend with different providers. To start, a project has to be created on their web platform after which API key is generated. Next, using their Python library, a session can be started simply by initializing the tool with provided API key at the beginning of the program and closed at the end of the program. Interactions and calls during the session will be recorded automatically and can be later analyzed on their web dashboard. [46]

In conclusion, very abstractly describing, the experimental environment consists of the running large language model, an external static application security testing solution for implementing a custom tool and a multi-agent system built in Python along with functionality to communicate, sync and modify files in the dataset running in Docker container.

4.2 Multi-Agent Framework

Multi-agent framework is a software environment that provides the tools to develop, coordinate and interact with multiple autonomous agents. This helps to save a lot of time when creating systems consisting of multiple agents. They provide a simple way to register and specify agents needed for certain purpose. This process includes describing the background, nature and goals for the agents. For example, if the goal is to analyze latest cybersecurity incidents, an agent with a background of being a senior cybersecurity analyst with 10-year experience could be created. Additionally, specific tasks with expected outcome have to be described. Majority of those frameworks take care of the process how agents interact with each other, manage their memory and knowledge base automatically and make it possible to provide agents different tools. [47]

The criteria set by author for choosing the suitable framework for this experiment are following: has to have a possibility to debug and analyze agents' outputs, communication and decisions, doesn't have big learning curve and should be usable through their official library or API using programming language, preferably Python. The ability to monitor agents' outputs and their interaction can give a lot of useful feedback on what the process is missing or what to improve. Using multiple agents in sequential order means that one agent's decision and answer depends strictly on the previous agent's answer. Therefore, if the end result is not in the form of what the user expects or is defective, without seeing what exactly is happening internally when the system tries to solve a problem, it is difficult to point out where the system acts incorrectly. Compared to just building an application based on multiple autonomous agents, this is even more important for experimental studies because one needs to see if and how every little change affects the behavior of the system. Often systems that are very scalable and highly customizable tend to be difficult to use and require time to work with them efficiently as they have to offer a lot of different functionalities. Since the purpose of this paper is not to build a complex business associated application, but rather a straightforward workflow for experimenting, scalability and detailed adaptation are not that important. Because of lack of previous experience with multi-agent frameworks, time limit and absence of specific requirement, the framework should rather be with a small learning curve and as transparent as possible. This will also make the buildable system easily understandable and simple to refactor and change the parts of it during exploration phase. Since handling and working with the dataset has to be done with the help of writing code, managing the multi-agent system should also be possible with programming. The no-code approach is getting more and more popular and the field of AI based applications is no different. However, in the context of this paper, the author prefers using the multi-agent building framework and tools through programming language as it provides more transparency and control over the system flow.

Although using multiple autonomous agents with artificial intelligence is relatively recent approach, there already exists multiple developed frameworks that have active support and community. There are some that are developed by smaller communities and also ones that have larger technical background, resources and are developed by huge tech corporations. For example, one of these is called AutoGen [48] that is developed and maintained by Microsoft. AutoGen is an open-source multi-agent AI application creating framework that is able to work alongside humans and act autonomously. It has Python libraries that make it possible to interact with the framework using code. Although it does have an independent .NET version of the framework, the main focus of the product seems to be on Python. The framework's core provides a simple way to build and launch scalable, event-driven AI agent systems. It contains features like multiple language support which currently are Python and Dotnet, being highly customizable by offering memory as a service, tools registry and custom agents and being easily scalable. Additionally, it provides asynchronous messaging between agents and a way to monitor and debug currently worked on agent systems. [48] They have a separate product called AgentChat which is a high-level API constructed with a goal of building multi-agent applications and is mainly aimed at beginners. It is built based on their main core package, but instead of advanced core programming model that offers the user more freedom and customization, the AgentChat is more abstract and therefore provides shortcuts to get started faster. [49] Data logging can be done in two different modes - SQLite database or file. Logging has to be started and ended manually by calling accordingly runtime logging starting and ending functions. [50]

Next framework candidate is called CrewAI that is not maintained by any big tech company, but rather a small team and was initially created by one person, João Moura [51]. CrewAI [34] is the leading multi-agent open-source platform that makes creating and orchestrating different AI agents simple. It has an independent platform with graphical user interface to manage and automate workflows using multiple agents which is more aimed at enterprises and is a paid product. Besides that, they have an open-source framework written in Python that is free to use and is the target for evaluation in the context of this paper. CrewAI is being used by multiple globally known companies like Oracle and KPMG, has over 28 000 stars on Github and is in use in over 40 percent of Fortune 500 companies. While some multi-agent frameworks are built on top of other larger frameworks like LangChain [52] then CrewAI is completely independent and built from scratch. It also offers two different ways to use the framework - first is called "Crews" that provides more abstract controls and higher-level customization. It enables user to create AI teams consisting of agents that each have their own goals, roles and tools. Its official tool selection includes tools for example for scraping elements from websites, extracting data from files supporting different formats and interpreting Python code [53]. Additionally, developer can build their own tools by

implementing custom logic and registering it as a framework tool. Agent tools are simply Python functions that contain description of its nature so the agent will know what and when to use it for, potentially inputs and its operating logic. Since agent will purely use the tool's output for its work, developer has a great freedom of choice deciding what the tool should do. Other approach of using CrewAI is called "Flows" which gives the developer opportunity to go deeper with detailing, providing making single large language model calls for extra precision and other granular controls. For debugging and monitoring the framework has verbose mode which provides insights on how agents execute internally just by toggling it on. [54] Reviewing its documentation, getting started with using the Python framework seems relatively simple - install dependencies, define agents with their tasks and the minimalistic program should already work. In addition to the seemingly small learning curve, it also has a very logical syntax and structure regarding different components of the framework and their connectivity. Defining agents and tasks can either be done directly through variables in Python file or externally using files in YAML format [55].

Third possible choice is LangGraph. LangGraph is a multi-agent framework that provides low-level control of the orchestration process. It offers long-term memory, customizable architectures and in case of handling complex tasks, human-in-the-loop interaction [56]. It is used by well-known companies like Elastic who uses it for an assistant as a threat detector and Uber who generates unit tests with the help of LangGraph. It is important not to confuse LangGraph with LangChain [52] as they have very similar names. Although they are different products, they are still closely connected. LangChain is more of a general product that provides high-level components and integrations for building large language model applications. LangGraph is a separate library built on top of LangChain and is therefore part of LangChain's ecosystem meaning it integrates effortlessly with other LangChain products. The name of the framework also expresses its working process - it is based on graphs meaning that components of the program are connected by nodes and edges. [56] It is also based on Python and has multiple libraries to be used to build multi-agent applications in that environment. According to the documentation, setting up the framework and launching primitive application with it isn't hard, but isn't very straightforward either. When wanting to go further, it seems that it can get quite complex and requires some basic or low-level components to be built by user himself. [57] Logging and debugging is solved through different streaming modes which can be attached to streams of graphs. There are different modes for streaming like values, messages and debug. According to the specified mode, corresponding info is output. [58]

In conclusion, three multi-agent frameworks were selected for evaluation: AutoGen [48], CrewAI [34] and LangGraph [56]. Main criteria for choosing the best suitable product were following: ease of monitoring and debugging agent interactions, minimal learning curve and small complexity and usability through code, preferably Python. Each criterion was broadly explained from the point of the framework in its description. Next, criteria matchings of chosen candidates are set side by side and finally the best one will be selected. Firstly, focusing on the possibility and simplicity to see and analyze agent's work process, AutoGen's solution seems the most uncomfortable since it provides logging either to database or file and has to be managed manually. CrewAI and LangGraph are similar in the case of monitoring as logging to console can be toggled with just one option. One difference is that LangGraph provides multiple modes for that, but those don't seem that important in the context of this paper. In terms of the learning curve, LangGraph has the steepest learning curve followed by AutoGen. CrewAI appears to be easier to adopt than the other two and has the most logical structure and setup. It is also fully independent and differs from others by not being part of another ecosystem. All candidates have the opportunity to be used with Python and have a strong support for it. Therefore, in this context one does not outweigh the other. Considering all the criteria, CrewAI looks to be the most suitable candidate of the three as it doesn't have a big learning curve, is well structured and has a simple logging functionality.

4.3 Dataset

There are several publicly available datasets that can be used in studying areas related to software vulnerabilities. Based on how they are built, they may include only code snippets of vulnerable functions or whole files and projects that contain a vulnerability. Some of them also contain additional information like CVE codes and official vulnerability fixing human developed patches. One of the criteria by which the author chose the dataset was that it should contain actual real-world vulnerabilities that are not artificially created for testing purposes. This means that the vulnerabilities have to be registered in some reputable vulnerability database and be linked with corresponding CVE code. This ensures that the large language model system has to handle the context and nature of real production-ready code. When using single snippets of artificially vulnerable functions, the result of the research may not indicate how the multi-agent LLM system acts when used on real projects. Second criteria for choosing the dataset was the capability to evaluate and validate the generated potentially vulnerability-free result. As multiple datasets include officially fixed code besides the vulnerable one, semantically comparing fixed code with the generated result is one option. This alone however does not ensure that the result of the LLM even compiles and therefore repairs the existing vulnerability. Therefore, the author looked for a capability to evaluate the results within the dataset. This could be accomplished with including test cases for each vulnerability or use some other alternative method to run and validate the result. Since the action of multi-agent system analyzing and processing vulnerable code is resourceful and takes time, the number of used records will be limited. Therefore, the large size of the dataset isn't among the first things in the prioritization list.

First filtered out datasets included DiverseVul [35], Big-Vul [36], Vul4J [31] and CVEFixes [59]. Although the selection for vulnerability repairing based datasets compared to simply code fixing datasets isn't big, there still exists multiple ones with different properties. Some of them are purely based on scraping large vulnerability databases and collecting different CVE-s along with vulnerable code. Others have additional information, such as officially applied patches for corresponding vulnerabilities and specially crafted tests to prove the existence of the vulnerability in the code.

DiverseVul is a vulnerable source code dataset that is constructed with a purpose of being used in deep learning based vulnerability detection [35]. This dataset contains almost 19 000 vulnerable functions that cover over 150 common weakness enumerations (CWE) and over 330 000 non-vulnerable functions. Compared to other prior similar datasets, DiverseVul is a lot bigger and can therefore provide a more realistic result. As it does include vulnerable code, but is more directed to the vulnerability detection research not vulnerability repairing research, it is a possible candidate, but rather a weak one. [35]

Big-Vul is a C and C++ programming language based vulnerability dataset that includes CVE codes and code changes [36]. This dataset is built by crawling different open-source projects and searching public vulnerability databases to link the found code with according CVE-s. Found vulnerabilities contain information like CVE ID-s, CVE summaries and CVE severity scores. It consists of over 3700 vulnerable functions across 91 different types of vulnerabilities that are extracted from almost 350 different publicly available Github projects. The whole dataset is formatted in a CSV format that makes it easy to analyze and process by another programs. [36]

Vul4J is a vulnerability dataset that consists of real-world Java code vulnerabilities [31]. It includes test cases for proving that the stated vulnerability actually exists in the code along with the official human patch and other information for reproducing the vulnerability. Currently it contains a total of 120 different vulnerabilities of which 79 have test cases. Additionally, besides the dataset, it includes a framework that allows the user checkout to a certain vulnerability, compile it and run the according test cases which makes managing and working with data easier and more understandable. [31]

CVEFixes [59] is a collection of vulnerabilities from different open-source projects along with provided fixes. It is constructed automatically by crawling and collecting the CVE records in the U.S National Vulnerability Database [60]. It covers over 12 000 different vulnerable code fixing commits from over 4200 projects. The dataset is built to act as a relational database making it easier for users to work with. They provide both the full dataset as well as the code for the automated vulnerability collection so that anyone can run the process themselves. [59]

4.3.1 Evaluating Datasets

Based on the first criterion, dataset containing real-world vulnerabilities, all of the previously mentioned datasets pass. All of them are constructed either by crawling large vulnerability databases, open-source projects or utilizing the combination of those processes. Looking from the angle of the convenience on working with the dataset, both Big-Vul and DiverseVul fall behind. Although DiverseVul is in JSON format and BigVul in CSV format, they don't have the extra functionality for handling the data like the other two - CVEFixes a relational database structure and Vul4J independent framework. While the database-like structure is easily understandable and workable with, upon analyzing Vul4J framework, it still can't beat that extra functionality. Vul4J framework's additional features makes it simpler to integrate it into automated experimental flow which is valuable in the context of this paper. Evaluating the ease and preparedness of validating experiment

results, Vul4J clearly stands out from other ones by having test cases. Although it contains less vulnerabilities than all the other ones, as said before, the size of the dataset isn't that important in this context. Therefore, Vul4J was the dataset author decided to proceed with as it matched best with provided criteria and had additional useful properties compared to others.

4.3.2 Chosen Dataset

Vul4J can be run on a host machine directly or in a Docker container. When choosing to run it directly on a host machine, the machine must run either on Linux or MacOS, have certain versions of Java, Maven and Python installed. Compatible versions of those dependencies are listed on their Github page [31]. For Docker image, there are two options available - latest image which is more lightweight and contains, as the name says, latest updates and "all-dependencies" option which is bigger in size and includes older compatible dependencies for all vulnerabilities to be reproducible. [44] Since some of the vulnerabilities rely on dependencies that are deprecated for now, reproducing those vulnerabilities didn't work anymore. The larger Docker image of the dataset addresses this issue and solves it by including all the required dependencies with compatible versions.

While some vulnerable code datasets contain only single functions or code snippets that are vulnerable, Vul4J [31] contains whole projects that are vulnerable. All vulnerabilities in the dataset have a unique ID in a format of Vul4J-ID which can be used to target a specific record. For each data record, there is a separate folder called "VUL4J" that contains human patch and a JSON file including a full path to a vulnerable file. The framework provides a functionality to checkout to a certain data record with the following command:

```
vul4j checkout --id VUL4J-5 -d /tmp/vul4j/VUL4J-5
```

This copies the ID equivalent project to the specified folder with additional files mentioned before. Now the vulnerable project is in its own folder and can be worked with individually. Next, one can compile the project to see if it compiles without errors and is successful. By default, all the projects should compile successfully, but this is useful when starting to fix the vulnerable file to ensure that it doesn't break the project. The compilation can also be done through the framework with the following command:

```
vul4j compile -d /tmp/vul4j/VUL4J-5
```


This eliminates the need for setting up an environment to compile and run the code yourself and therefore also excludes potential problems that might arise from having different environment properties. As mentioned before, the majority of dataset records have test cases proving the presence of vulnerable code. After the vulnerability has been compiled, it is possible to run those tests and get summary of results in JSON formatted file with the following command:

```
vul4j test -d /tmp/vul4j/VUL4J-5
```

Test results include information like vulnerability ID, CVE code, repository with a link pointing to the origin project and a link to the human patch commit and statistics for tests. Statistics include the number of tests that were run, number of ones passing, number of errors and lastly the number of tests that failed or were skipped. Additionally, it lists exactly which tests passed, which were skipped and if there were any that failed, it outputs exactly what was the test named and why did it fail. The framework also enables user to reproduce already existing vulnerabilities that are listed in the CSV dataset as well as new ones that the user can add themselves. The process of reproducing vulnerability includes compiling it, running specified tests and analyzing with the built-in SAST tool. If there are no tests or warning specified in the CSV dataset, those steps are skipped. [31]

4.4 Selection of Large Language Model

Other similar studies [6] [32] that focus on vulnerable code repair with the help of large language models use ones that either have the flagship status, like GPT-4 [30] or are specifically fine-tuned for this purpose. Recent rapid development of new LLM-s like DeepSeek R1 [61] and the publishing of new models' training and development techniques lays a promising base for smaller publicly available models. Therefore, this paper is mainly focusing on using two differently sized large language models by DeepSeek [61] - DeepSeek R1 with 32 billion and 671 billion parameters. For initial development of a working workflow and analysis of datasets, smaller LLM-s provided by Ollama [37] are used. These will run on the author's personal computer and are used through LAN on another computer where the workflow is being developed. Smaller sized model used to generate final results of the experiment is run on TalTech's infrastructure and bigger model is deployed and used through Azure AI Foundry platform [62]. Both of them are communicated with through their API-s accordingly.

Choosing the DeepSeek R1 model [61] as the base model over other available models was motivated by multiple key factors. Firstly, since the R1 is a fairly new model in the context of being publicly available, being released in January 2025 [63], there aren't many studies published on it. Its release caused a lot of discussion about how much cheaper it was built compared to other similar sized models and its surprising performance in various areas. Secondly, related to the first point, R1 demonstrated competitive performance compared to other superior models. For example, it outperforms or performs very similarly compared to OpenAI model o1 [64] in areas like math or coding. Both models were tested on a mathematical benchmark consisting of complex mathematical problems where the DeepSeek's R1 model received 97.3% success rate and OpenAI o1 96.4%. They were also evaluated in programming capability with using them in a competitive programming platform that mirror real-world software development scenarios. In programming, R1 reached a success rate of 96.3% while the o1 model was slightly better and resulted in a 96.6% success rate. [65] Both coding and math experiment show less than 1% difference, making the DeepSeek's R1 large language model a good candidate for further studying and comparing in various fields. Lastly, it is simple to run smaller distilled versions of the original R1 model through providers like Ollama [37]. This enabled to explore how does the model size affect the results in a automated vulnerability repair process.

Additionally, DeepSeek’s price list is a lot of cheaper than for example OpenAI o1’s. Using R1 to output one million tokens costs 2.19 USD dollars according to its standard price [66] while the same process for o1 costs 60 USD dollars [67]. Figure 3 [63] illustrates the API input and output pricing for DeepSeek R1 and OpenAI o1 models considering both cache hits and misses. That is a significant difference between models that perform relatively similar in multiple fields.

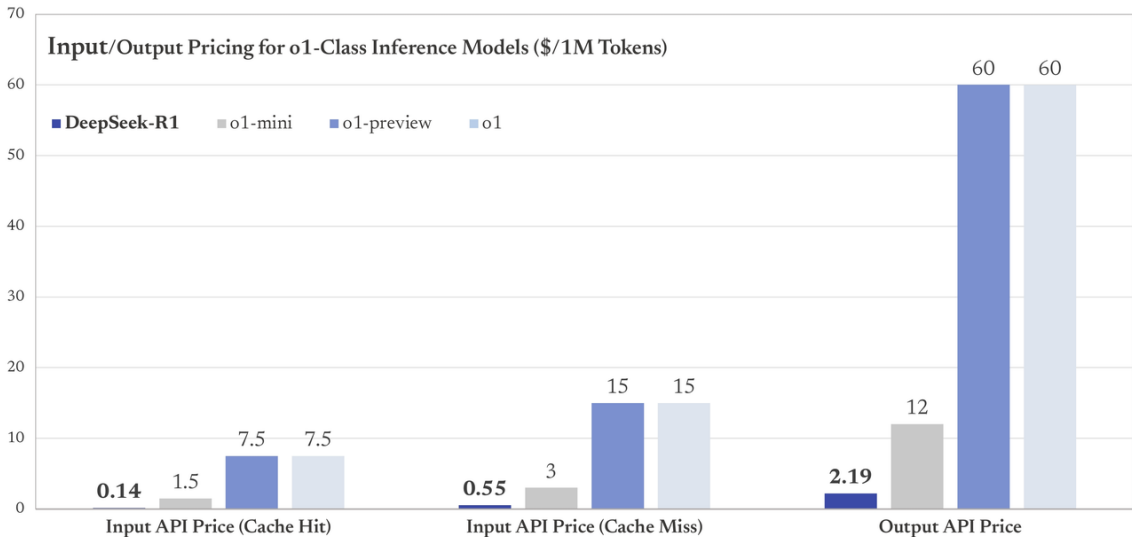


Figure 3. Price comparison of DeepSeek R1 and OpenAI o1

TalTech has an independent AI laboratory [68] primarily aimed for students to learn how to use AI to solve various problems. Students of TalTech who have given access, can SSH into the server using their official university account credentials and use the available resources. There are a total of 7 laboratories, each equipped with hardware of different capabilities. For example, there is a lab equipped with Intel E5-1620 CPU, 64GB of RAM and NVidia GTX1080Ti graphics card and another lab equipped with AMD Threadripper 3970X CPU, 128GB of RAM and NVidia RTX4090 graphics card with 24GB of memory. Thanks to this, users can select and checkout to a laboratory that meets the requirements for their experiments and testings. [68] According to the hardware requirements declared here [69], DeepSeek’s distilled R1 model with 32 billion parameters can be run with RTX4090 GPU. This confirmed the author’s plan to use TalTech’s AI lab to experiment with the distilled model.

Azure provides a platform called Azure Foundry AI [62] which enables to conduct AI operations, build and tune models and develop applications. It is specially designed for developers to build different AI based applications and explore, test and use AI tools. The platform offers wide variety of large language models that an user can deploy and start

using right away. Although navigating through Azure's different products and portals might confuse a first-time user, deploying and setting up LLM in Foundry AI is relatively simple. To start, a project has to be created. After defining and initializing the project, user has to navigate to the model catalog, choose which model they would like to deploy and click a button to deploy it. Next, an API key and specific URL is generated which, after the successful deployment, can be used to communicate with selected LLM. Additionally, Microsoft provides students credits and discounts in Azure to experiment with and try their products. In summary, all three of following aspects affected choosing the Azure Foundry AI platform to deploy and use original sized DeepSeek R1 [61] large language model - Microsoft product's reputation, ease of system setup and student benefits. [62]

5. Experimental Flow

Since evaluating multi-agent system efficiency in repairing vulnerable code relies on quantitative research method, it includes handling and processing a large amount of data. To make it efficient and less time consuming to experiment with, author decided to create an automated workflow starting from extracting the vulnerable code from data records to compiling and running tests on the possibly fixed version of the code. As the experimenting includes a lot of changing attributes and configuration to reach the best possible result, same processes have to be run multiple times. To make modifying the flow easier, it was built with additional functionalities like targeting or excluding a certain list of vulnerabilities by ID-s and structured so that intermediate steps can be excluded without breaking the overall process. An automated workflow makes both the testing phase as well as generating final results simpler, more flexible and timesaving.

5.1 Multi-Agent Setup

As stated above, a framework called CrewAI [34] is used to define different agents along with their tasks and generally handle the agent communication and activity process. It allows wide range of different customizable attributes to modify agents' life cycle. This includes modifying how tasks and agents are handled - either sequentially or hierarchically and is the planning enabled for pre-execution strategy. Initially, only two agents were used - software developer and security analyst. Analyst's goal was to analyze the vulnerable code, find a vulnerability and provide a detailed description with recommendations on how to fix it. Software developer got the information from the security analyst and tried to fix the code according to that. This agent setup was used to ensure that the agents were initialized correctly and that the system was working without problems. Since looping through the vulnerabilities and letting multi-agent system repair them takes a lot of time and resources, it is wise to test first on a smaller scale. The final setup used to conduct full-scale experiments consisted of 5 different agents - security researcher, security analyst, software developer, code sanitization engineer and code integrity engineer. Last two agents' roles in code aren't exactly the same as their names, but are more general to ensure that the system understands their background and specialization correctly. Therefore, their names in the paper are derived from their tasks and goals rather than their roles.

Researcher Agent

Researcher agent's full role description is cybersecurity researcher whose goal is find information and details about the currently processed vulnerability. It is provided the corresponding CVE code of the security flaw which it can use as a main identifier. It also has two similar web searching tools that it can use whenever it decides to - specific website scraping tool and more generic web searching tool. Its task is to find as detailed and useful information about the vulnerability as possible and construct a summary of it including information on how to repair it in the Java code.

Security Analyst Agent

Next in the sequence is a senior security analyst whose purpose is to spot the exact location in the code where the vulnerability lies and, with the help of information about the CVE from the previous agent, provide a thorough description of the situation and useful tips for developing and implementing a fix for it. The agent is described to have an extensive knowledge in cybersecurity and is provided both the vulnerable code to analyze and also the CVE identifier that the code contains. It is also provided a custom static application security testing tool that it can prompt to help it examine the code better.

Software Developer Agent

After the analyst comes the software developer agent who carries the main role in this team. It is described as a senior developer who is specialized in Java development and also has an extensive background in cybersecurity. Its goal is to analyze the previously described and located vulnerability and develop and implement a fix for it while keeping the rest of the code untouched. Additionally, to the CVE identifier and vulnerable code, this agent is also provided a vulnerable file name to specify what the class name should be. The developer agent is also provided some extra tips it has to follow. Examples of these are that it should not change Java class' inheritance properties or constructor unless really needed, it should not delete or rewrite any existing code that is not related to the vulnerability and it should take into account previously provided recommendations and information about the security flaw. Expected output of this agent is a repaired vulnerability-free version of the inputted Java file that compiles without errors.

Code Sanitization and Code Integrity Agents

Now that potentially non-vulnerable Java code is generated, it should also be correct semantically and compatible with the project it was extracted from. That is also the point why previous agent was told to modify only the vulnerable part of the code and leave the rest of the file untouched if possible. To ensure that the generated result does not have semantic errors, two extra agents are used. Penultimate agent's task is to clear the now repaired code of comments, explanations and other additional information so the end result

would be clear and pure Java code. Last agent's purpose in the sequence is to add all methods from the original file that were not associated with the vulnerability. This should help to raise the chance that the absolute end result contains a fix for the vulnerability as well as compiles within the project it originates from. Complete code semantic descriptions of the agents and their tasks can be found in appendices. Figure 4 abstractly illustrates the process of agents' interaction with each other and LLM to producing a final pure code result.

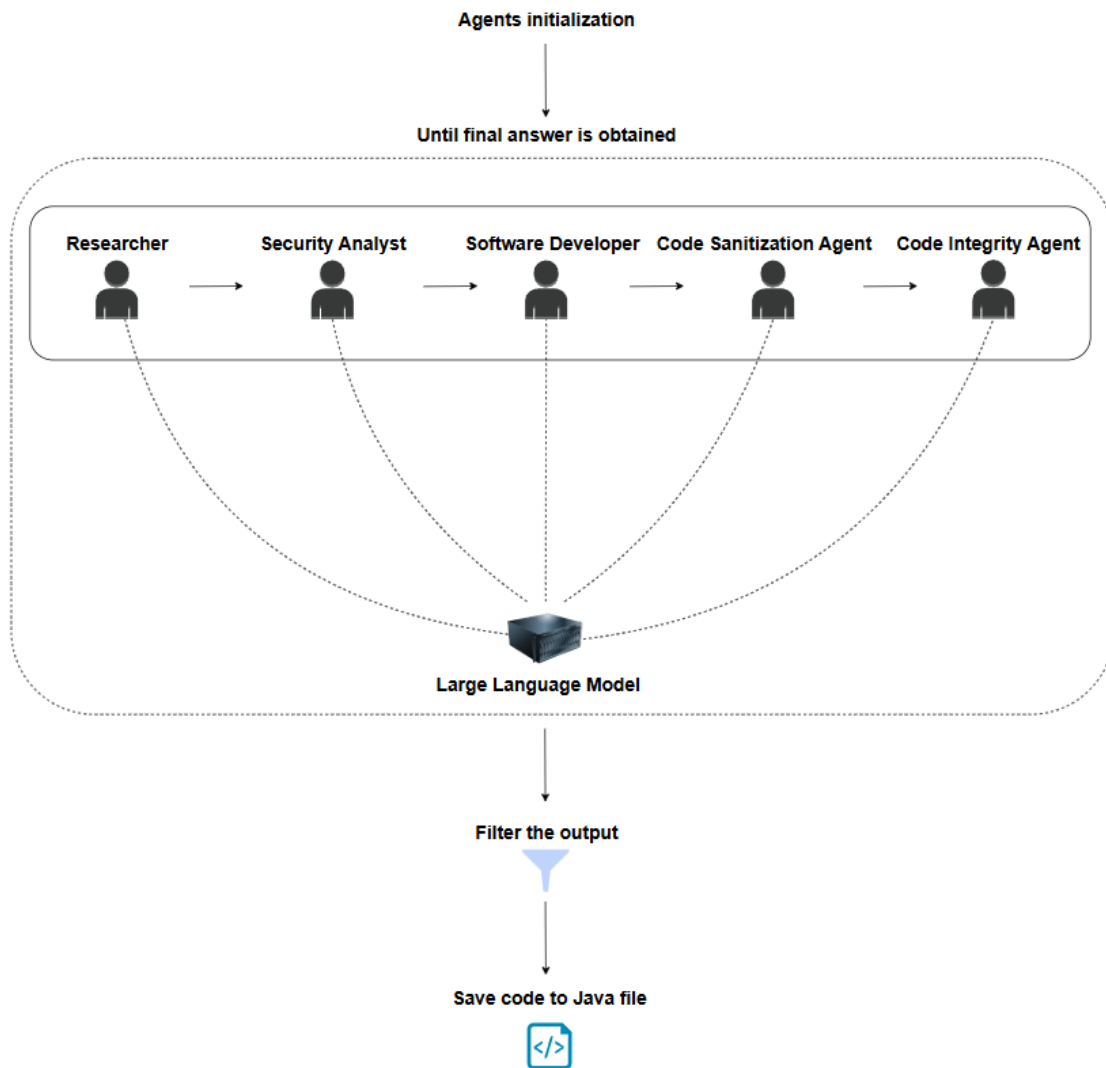


Figure 4. Abstract process of agents' interaction with each other and LLM to resulting final Java file

As mentioned in agents describing section, some of them also have access to different tools. In the context of this paper, three different tools are used. First of them is from CrewAI's [53] official tool selection that can be imported from their independent tool Python library called *crewai_tools*. After that, it can simply be initialized, configured to user's needs in the code and passed to the desired agent. Name of the tool is SerperDevTool that is built to search across the internet and find most relevant results for the topic searched [70]. It is based on another provider's, Serper [71], API that claims to be world's fastest and cheapest Google search API. Using this, the tool performs a semantic search with provided query, fetches results through the mentioned API and returns them. As it uses an external party's service for its job, a freely retrievable Serper API key has to be defined as an additional step. It has multiple configuration options like the endpoint for search API which by default is *https://google.serper.dev/search*, specification for country or location to search for results and a number of search results to return. In conclusion, this tool allows the system to conduct relevant searches about specified topic. The next used tool is similar in a sense that it also focuses on getting data from the web. It is also from CrewAI's provided tool list and is called ScrapeWebsiteTool. Unlike the previous tool, this one deals with extracting information from a specific website rather than using a search engine to find results. It is specifically designed to read and extract the content from a provided website. As there exist websites that can be targeted by CVE identifier that include information about the specified vulnerability, it is a promising utility in this research. It works by making HTTP requests and parsing the received HTML content from the response. It only takes in one argument, website URL, which is mandatory because that defines the site that is being processed. [72] The last tool is not provided by CrewAI, but rather custom built by the author. It is a static application security testing tool that is based on another CLI tool called Bearer [39]. The working principle of it is simple - it takes in the dataset vulnerability identifier that is used to locate the vulnerable project. Then the Bearer tool command for scanning is executed on the project folder and result of this is returned to the agent. Further explanation of this tool setup and usage is described in the experimental environment section. Ultimately, as understanding and locating the vulnerability in the code are critical processes in fixing the security flaw, these tools should potentially enhance both of these procedures.

5.2 Filtering the Dataset

As stated above, the used dataset Vul4J [31] contains a total of 79 vulnerabilities that are equipped with tests. Some of those vulnerable projects have multiple vulnerable files. Study with similar focus [32] using the same dataset used only single-hunk vulnerabilities. Single-hunk vulnerability means that the vulnerability exists in a single code block or single line. Reasoning for choosing these as a target group was that the APR models that they investigated were built to target single-hunk bugs. [32] To avoid unnecessary complexity in building LLM prompts and handling results, the author also decided to exclude multi-file vulnerabilities from the targeted experiment dataset. But for more realistic scenarios, instead of focusing on single-hunk vulnerabilities, author decided to target single file vulnerabilities meaning that the security flaw could be spread across multiple lines, but must be in a scope of a single file. In the dataset, there are 11 records that contain multiple files containing security flaw. Therefore, all of the vulnerable projects with more than one vulnerable file were excluded. Before starting to actually repair and run tests, author decided to compile all of the projects as they are in the dataset. As a result of this, it appeared that there exist some projects that fail to compile by default. There were 14 of such projects. These were also excluded from the final target group as they would have resulted in false positives. Since most of the large language models have an input size limit which can be forwarded to them, two of the vulnerable files combined with agent's prompt were too long. Because it is not that critical for a dataset to be big in the context of this research, these two vulnerabilities were also excluded. Finally, starting from the original amount of 79, excluding 11 of the ones with multiple vulnerable files, 14 of those that did not compile, one that the LLM was unable to process and two that were too long resulted in a working dataset containing 51 vulnerabilities. To clarify what are the natures of the filtered vulnerabilities, author grouped them by Common Weakness Enumerations. Table 1 below displays each Vul4J [31] vulnerability used in this study being mapped into correct CWE category along with the enumeration's short description. All of the CWE descriptions are derived from the official CWE Mitre website [73].

CWE code with description	Vul4J ID
CWE-20 - Improper Input Validation	1, 10, 26, 30, 48, 49, 66
CWE-22 - Path Traversal	18, 41, 43, 65, 69, 76
CWE-74 - Injection	45
CWE-77 - Command Injection	33
CWE-79 - Cross-site Scripting	25, 34, 50, 59, 60
CWE-264 - Permissions, Privileges, and Access Controls	28, 29
CWE-269 - Improper Privilege Management	52
CWE-284 - Improper Access Control	22
CWE-287 - Improper Authentication	40
CWE-310 - Cryptographic Issues	44
CWE-502 - Deserialization of Untrusted Data	54, 77
CWE-532 - Insertion of Sensitive Information into Log File	57
CWE-611 - Improper Restriction of XML External Entity Reference	2, 24, 47, 61, 64
CWE-835 - Infinite Loop	6, 7, 8, 12, 13, 53, 55
Not Mapped	5, 9, 17, 19, 20, 31, 32, 46, 70, 71

Table 1. Used Vul4J vulnerabilities grouped by CWE categorizations

As seen from the Table 1 above, excluding the ones that aren't mapped to any CWE, the highest number of vulnerabilities are equally divided into two categories - CWE-20 and CWE-835 which stand for improper input validation and infinite loop vulnerability respectively with both containing 7 instances. These are closely followed by CWE-22 indicating the security flaw of path traversal with having 6 vulnerabilities. Lastly, there are two categories which stand out with having larger number of vulnerabilities being CWE-79 and CWE-611 which point to cross-site scripting and improper restriction of XML external entity reference respectively. They both contain 5 vulnerabilities each making them last groups having a greater weight compared to other CWE-s. Failing compilation of some vulnerabilities was probably caused by deprecations of different dependencies and libraries. Therefore, at least some of them, could probably have gotten fixed by using Vul4J full version Docker image [44] instead of the lightweight one. But since author would have anyway filtered the dataset to be smaller due to various reasons, including manual evaluation, it was decided to use the original image and avoid potential accompanying problems with additional features and more data.

5.3 Automated Workflow Description

In the context of this paper, Vul4J dataset [31] is being run in the Docker [43] container. Therefore, the first step is starting the container with the dataset containing folder synced to the host machine. Since the multi-agent system is not running in the same environment as the dataset, mirroring the data is essential. Since the commands related to dataset framework, for example for compiling the project and running tests, have to be executed inside the container, a connection with the specific running container has to be established. Once the connection has been established, preparing vulnerabilities for experiments can start. As the vulnerabilities can be identified with a specifically formatted ID, an array is constructed containing desired range of identifiers. By default, all of the previously filtered out 51 vulnerabilities in the dataset are targeted. However, since targeting specific vulnerabilities separately is necessary in various experimental scenarios, the program has the functionality that supports it. This list is then being looped through and every selected vulnerability is handled independently. For each data record, framework commands that checkout to a vulnerability and compile are executed. As a result of this, vulnerable project will be created, compiled and moved into a separate folder named after the vulnerability ID. In addition to the compiled project, this folder also contains a separate folder containing extra contents about the existing vulnerability. For example, it includes JSON formatted file that contains following information: CVE ID, official project URL, command for running tests, information on human patch including its GitHub URL with fixing commit hash, file path and contents. There are also two folders - one for vulnerable file and one for the patched file that contain corresponding Java file and a JSON file that specifies the file name and its path in the project. Using the vulnerable file path, vulnerable code is extracted from the correct file and set ready to be analyzed.

Next, a vulnerable Java file name, that is currently being worked on, is extracted. It is used in agents' prompts to specify what should be the name of the Java class for the end result and also for defining the file name where generated result is written in. Depending on the current experiment run configuration if agent tools are used or not, preparatory work for them is done. Three tools are being used - one for performing semantic search on the internet [70], second for reading and extracting content from a specified website [72] and last is a custom SAST tool. Therefore, an URL of the scrapable website have to be defined. For the website used as a base site for scraping information, a vulnerability specific NIST national vulnerability database website [60] is used. The URL structure of the website is the following: <https://nvd.nist.gov/vuln/detail/{CVEID}>. As the website's address pattern is quite generic, the CVE code of the vulnerability can

be extracted from the file mentioned above and appended directly to the NIST URL. This site provides detailed information about corresponding vulnerability, including description of the CVE, metrics along with base score, CWE categorization and links to solutions and advisories related to the CVE. Semantic searching web tool is not passed any argument directly, but the associated vulnerability extracted CVE identifier is passed to the multi-agent system and from there to the research agent. Therefore, it is still indirectly forwarded to the tool as the agent, with high chance, uses it for researching the security flaw.

5.4 Evaluation Method

To assess the effectiveness of the multi-agent large language model system for automated vulnerable code repair, a combination of manual verification and using unit tests was used. The initial idea was to mainly rely and focus on using the dataset's tests as the ground truth for results verification. While experimenting with a small fraction of the dataset, the author observed that LLM-s struggled to generate the file with similar structure and essential attributes as the original Java file. Therefore, there were multiple scenarios where the multi-agent system actually fixed the vulnerability or provided a very similar solution as the official human patch, but was initially marked as failed because it did not compile. For example, if the generated code contained the same security flaw fixing line at the same location as the human patch, but imported an exception originating from a library that the project didn't include, the tests were not executed. That is one of the reasons why manual verification was essential in the evaluation process.

A total of four different categories were created to label results. Large language model outputs are inherently generative and the model itself is not bound to follow any fixed structure in its response. Therefore, there are a lot of different ways a LLM can approach fixing the vulnerability which often makes it hard to evaluate the generated solutions using binary classifications - complete or failed. Hence two additional segments were created. All of the evaluation categories along with their descriptions and criteria are listed below.

Complete The generated fix correctly addresses the vulnerability with a solution matching the ground truth patch. Minor modifications like adding an import or removing singular line are allowed. The vulnerability must be fully mitigated.

Highly Relevant The generated fix spots the correct location in code and implements a valid and logically sound solution that differs from the ground truth patch. Complete mitigation of the vulnerability is not required, but the solution must correctly address the issue and be meaningful.

Minimally Relevant The fix identifies the correct vulnerability and location in code, but fails to fix it or implements an invalid logic.

Failed If the generated result does not target correct location in code, does not implement anything similar to the actual patch or does not modify anything in the file.

If the generated fix is correct, but includes additional code in other locations that is not entirely irrelevant, it is still considered a valid fix. Since the LLM system is given a complete content of a Java file, it would be too harsh to discard a correct solution with minor unrelated modifications. Same evaluation logic is applied when the generated result is not compilable within its origin project, meaning it might not include one or more unaffected functions and instead contain a comment such as *Additional methods omitted for clarity*. If the fix for the security flaw is valid and there are no major or breaking changes made to the rest of the code, it is still labeled as complete or as one of the relevant categories, depending on the implementation.

5.4.1 Unit Tests

Initially, all the projects were compiled with the vulnerable Java file being replaced with the patched one. Purpose of this process was to ensure that test return correct results by default and that the human patch indeed fixes the vulnerability. This resulted in JSON formatted files containing default test results for every vulnerability. This was done to later compare generated solutions tests results with official ones. Prerequisite for running tests is that the project must compile successfully with having the original exploitable file replaced with the generated one. Therefore, as stated in the section describing the experimental workflow, tests were automatically executed only when the project compiled without errors. After all test results of both generated solutions and original patched projects were acquired, the numbers of failing, skipping, running and passing tests were compared.

5.4.2 Manual Evaluation

Each vulnerability repair, not depending on whether it compiled or not, was manually inspected. The generated result was first compared with original vulnerable file to determine if the file is in correct format and if any changes were made. If the generated code was completely identical to the original file, it could be right away marked as failed without further analysis. Depending on the structure of the LLM outputted file, similar logic could be applied. Files consisting of only singular function without any attributes of Java class structure, unless perfectly fixing the vulnerability, were also immediately marked as failed. If the generated result implemented correct attributes and structure of original class, but failed to include some non-vulnerable methods that the original contained, the result was allowed for further assessment. Results that passed the first phase were then compared

against the official human written patch. Depending on the final evaluation, the large language model system generated solution was then put in one of the following categories that are described above - complete, highly relevant, minimally relevant or failed. Figure 5 illustrates the whole described manual evaluation process.

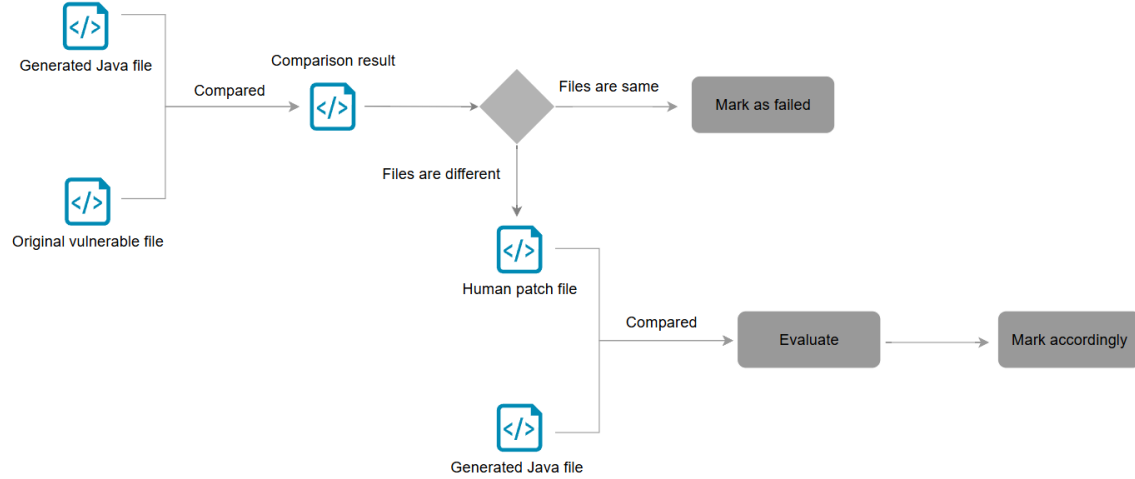


Figure 5. Process of manual evaluation of the generated result

A closer look at the unit tests revealed that some of these were constructed to depend on attributes irrelevant to the vulnerability. By that it is meant that certain tests checked specific variable values or thrown exceptions to determine if the test passes or not. However, this approach did not always confirm or rule out the existence of the vulnerability in the code. For example, a vulnerability with the identifier of VUL4J-49 [31] has its tests based on a fact that upon executing a certain function, an exception instance of *InvalidJWTSignatureException* class is expected to be thrown. In preliminary experimental results, the system was able to fix the vulnerability in a correct way, but threw a *InvalidJWTException* class exception instead. Therefore, tests results did not indicate the real state of a file regarding vulnerability. Considering similarly constructed tests and a requirement of successful compilation to even run tests, author decided to shift the main focus in evaluation to manual verification and use tests as an additional method.

6. Results

The results of this study were obtained through multiple experimental runs using two different sized DeepSeek's [61] models - DeepSeek R1 671B and DeepSeek R1 32B. Reason for utilizing the smaller model was to show the difference in performance resulting from the model size. Both models were used to conduct runs with three different configurations. First run was done using the five-agent setup described in the multi-agent setup section above. In addition to the agents' goals and tasks, the system was provided the complete source code of the vulnerable file and the file's name. Providing the Java file name was done to enhance the chances of the generated result being similar to the original file so it would compile in its origin project. The second run was very similar to the first one with a difference of having additional information on the vulnerability and access to different tools. Extra information included CVE code of the specific vulnerability and tools provided were two web scraping and searching tools and one SAST tool. This made it possible for agents to search information about the particular security flaw from the internet and analyze the code for potential weak spots. Final run was done utilizing the traditional approach of using a single prompt to communicate with LLM. Similarly to the second iteration of the experiment, the prompt also included CVE code, vulnerable file source code and file name. The purpose of this run was to set a baseline to compare multi-agent solution results to. All of the results are shown in Table 2 displayed below. First run's results are marked as "Multi-agent w/o tools & hint" in table, second run's results as "Multi-agent with tools & hint" and final run's as "Without agents and hint". Hint in this context means vulnerability specific CVE code. Complete, highly relevant, minimally relevant and failed are marked in the table as follows: Comp., H. Rel, M. Rel and Fail.

Table 2. Comparison of results using different configurations with two models

Model / Approach	DeepSeek-R1 671B				DeepSeek-R1 32B			
	Comp.	H. Rel.	M. Rel.	Fail.	Comp.	H. Rel.	M. Rel.	Fail.
Without agents and with hint	6	9	3	33	0	2	0	49
Multi-agent w/o tools & hint	5	7	6	33	0	1	1	49
Multi-agent with tools & hint	11	7	6	27	1	2	1	47

A total of 51 vulnerabilities from the Vul4J [31] dataset were used. As expected, the best result came from using DeepSeek’s R1 [61] full-size model with 671 billion attributes accompanied with the system having information about the vulnerability and access to tools. It managed to completely fix 11 vulnerabilities that were very similar or exactly the same as the human patched version and required minor modifications. In addition to complete fixes, this setup managed to provide 7 solutions that were categorized as highly relevant and 6 as minimally relevant. Next iteration was conducted using the same setup except without hint and agents’ tools. This resulted in 5 instances being classified as completely repaired and 13 as relevant, with 7 of them categorized as highly relevant and 6 as minimally relevant. Last iteration using the full-sized model done using the traditional approach with hint included successfully repaired 6 vulnerabilities and provided relevant code for 12 of them of which 9 were above average relevant and 3 which had abstract hints or implementations of the fix. Running the same iterations using the distilled model produced following results: 1 successfully repaired and 3 relevant fixes using multiple agents with tools and hint, 2 relevant fixes with same setup, but without tools and hints and 2 relevant fixes using the traditional approach. Distribution of the relevant fixes between two levels were following: 2 highly relevant and 1 minimally relevant with tools and hints, equally 1 minimally and 1 highly relevant without tools and 2 highly relevant using no agents.

6.1 Validating Using Tests

As described in the evaluation section, Vul4J dataset [31] tests do not always express if the file is vulnerable to the specific security flaw or not. This is because certain tests check for specific exceptions to be thrown or word-by-word exact same strings to be returned in order to be passed. This often resulted in a situation where the vulnerability was actually mitigated, but tests still failed. Due to this kind construction of tests, they were used rather as an additional verification method. Every generated vulnerability fix that was categorized as completely fixed using multi-agent system with tools and hint utilizing full-sized R1 LLM was verified by running tests. Table 3 presents a summary of the results, listing each vulnerability separately that was marked as completely fixed in the manual evaluation process. Along with the vulnerability identifier, an additional comment is included about the required modification the user needed to make in order for the solution to pass all its tests. Empty slot means that complete solution was not generated by the specified configuration. The table uses a set of descriptive markings to indicate required actions. The "+" sign means that the solution was correct as it was generated and needed no further modifications. *Remove Single Line* indicates the need of deleting a singular, often unrelated to vulnerability, additional line of code. *Relocate Single Line* marks the requirement of moving one line of code which in the context of this experiment was an if-clause in both scenarios. *Add Import* signifies the addition of a single dependency and *Append Methods* the need of appending non-vulnerable methods from the original file for successful compilation.

Table 3. Required modifications to pass Vul4J tests

ID	Multi-Agent w/ tools and hint	Multi-Agent w/o tools and hint	Traditional w/ hint
VUL4J-13	+		
VUL4J-41	+	+	
VUL4J-43	+	+	
VUL4J-45	Remove Single Line		
VUL4J-46			Append Methods + Remove Single Line
VUL4J-47	Append Methods		Append Methods
VUL4J-49		Relocate Single Line	Relocate Single Line
VUL4J-50	+		
VUL4J-57			+
VUL4J-61	Add Import	Remove Single Line	
VUL4J-64	Remove Single Line	Add Import	
VUL4J-66	+		
VUL4J-69			+
VUL4J-71	Add Import		
VUL4J-77	+		+

As seen from the table, the proportion of the returned solutions that categorize as a fully compatible Java file, meaning they didn't need any modifications, drops gradually going setup by setup. The highest percentage of 54.5% comes from the setup with tools and hint which is followed by the configuration without tools and hint resulting in exactly 50%. Configuration without agents reached a 40% in providing fully compatible fixes. Percentage wise, the run done without agents produced significantly more incomplete files which required appending methods from the original file to be compilable. There were three vulnerabilities that got all fixed by at least two different setup runs and didn't need any modifications in any case - VUL4J-41, VUL4J-43 and VUL4J-77. There were some exceptions for vulnerabilities with ID-s of VUL4J-13, VUL4J-41, VUL4J-43 and VUL4J-57 where tests expected an exception to be thrown with certain text. Although these tests failed initially because the message was not correct word-by-word, it was marked as successful because the vulnerability was mitigated. As for the completely fixed vulnerabilities' natures, the most fixed vulnerabilities belonged to common weakness enumerations CWE-22 and CWE-611. CWE-22 marks path traversal vulnerabilities and CWE-611 points to the improper restriction of XML external entity reference flaw [73]. Vulnerabilities belonging to CWE-22 category, including previously mentioned VUL4J-41 and VUL4J-43, were repaired 5 times and flaws belonging to CWE-611 6 times in total across all three configurations. To confirm this evaluation, all solutions marked as highly relevant were also grouped by CWE-s. Looking at the distribution of highly relevant results across CWE-s, the credibility of the first assumption done on completely fixed solutions is correct as the relevant solutions included 4 CWE-611 and CWE-22 vulnerabilities. This is the most amount of instances in one category which is followed by having 2 fixed vulnerabilities in CWE-79 and CWE-74 enumerations. In conclusion, majority of LLM produced solutions were compatible and passed tests in their origin form or required very minor modifications like adding a singular import or removing vulnerability unrelated singular added line. This confirmed author's markings about the vulnerabilities specified in the table above.

6.2 Usage of Provided Tools

For tracking the usage of a provided custom SAST tool, an indicator was created for each execution. Every time an agent decided to use this tool, a file was created into a folder named after the specific CVE currently being handled. Since agents act autonomously and make their decisions themselves based on the context they have, they can't explicitly be forced to use certain tools. Therefore, access to tools can be given, but they still have to be initialized by agents independently. It was observed that during the run using full-sized model and tools, the system used the SAST tool for 16 out of 51 vulnerabilities. However,

when provided the same tool for an iteration done using distilled model, none of the tool executions were recorded. This indicates that smaller models are not that efficient using additional tools in its work. Online discussions and articles [74] confirm the author's hypothesis that the usage of tools strongly depends on the capabilities of the used LLM. Large language models, especially ones with fewer parameters, might sometimes pretend that they are calling the tool, but are actually not and instead hallucinate by making up the output itself. Additional reason for tools not being used are that some LLM-s are trained to be too focused on specific context and are therefore limited for these opportunities, meaning that during its thought process, it does not reach a decision of calling or even discovering tools it is provided. [74] Although two other tools performing web searching and scraping were not individually tracked, based on previous information, a similar hypothesis can be made. This is also reflected in the results using a distilled model where the difference with tools and without tools are only one complete and one relevant fix as the same situation using a full-sized model results in a much bigger gap.

An AI analysis and monitoring tool called AgentOps [46] whose description and implementation process is described in experimental environment section was used. In the context of this study, it was mainly used to help monitoring agents' communication with each other and also them interacting with provided tools. The platform displays a whole timeline of a specific recorded session that is divided into different sections. Figure 6 displays how it looks visually on the platform. Its purpose is to visually represent the chain of events consisting of agents and tool interactions.

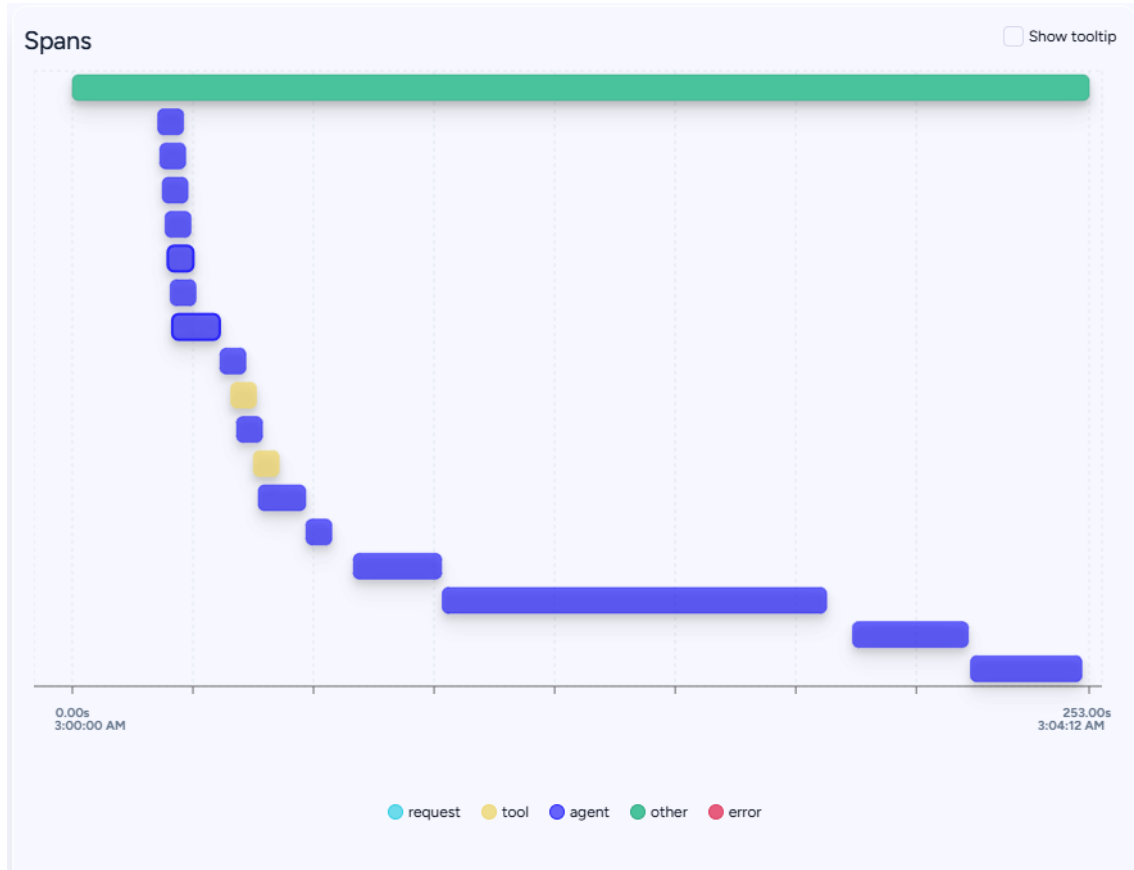


Figure 6. LLM session divided into different states

Blue lines indicate interactions with the large language model. These include both user specified agents that in the scope of this study are for example software developer and security analyst, but also intermediate actions done by the framework like the team planning. Short yellow lines display agents' interactions with tools. Although observing the exact tool section of the session does not expose any details about it, analyzing the next agent call reveals the usage and results of using the tool. Looking at the steps made in a session using the bigger model, it can be seen where the system agent decides to use both web searching and scraping tools. After listing its complete plan how it decides to execute

the process of repairing vulnerable code, the next thing it does is use a web scraping tool to get information about the provided CVE. Figure 7 below displays the first response of a LLM after the agents had been initialized and their steps had been planned.

```
<think> Okay, let's tackle this. The user needs detailed info
on CVE-2017-18349 to fix it. First, I should check the NVD
page using the Read website content tool. That's the primary
source. Let me do that.
Action: Read website content Action Input: {}
```




Figure 7. Decision indicating web scraping agent tool usage

It shows that the system plans to use the exact website, vulnerability specific NVD NIST page [60], that the tool was configured to use beforehand which is described in the experimental flow section. This is exactly how the system flow was visualized when choosing and configuring tools to include. This is followed by the tool execution step after which its result can be seen. The next agent interaction includes navigation headers of the NIST webpage and also vulnerability specific information that was present on the website. The described situation is presented as Figure 8 below.

```
| About NIST | Careers | News | Events | Publications | A-Z Index | Computer Security Division | Computer
Security Resource Center | Contact Us |
Hmm, the NVD page says that Fastjson before 1.2.25 has a remote code execution vulnerability via crafted
JSON. The issue is in the parseObject function. Pippo 1.11.0 uses FastjsonEngine, which is affected. The attack
vector is a crafted rmi:// URI in dataSourceName sent to /json URI in Pippo. The solution is to upgrade
Fastjson to 1.2.25 or later. The CWE is Improper Input Validation. Also, references point to GitHub issues and
security updates.
Wait, maybe I should check the GitHub links for more details. Let me use the Search tool to find more info,
maybe the advisory or fixes.
Action: Search the internet with Serper Action Input: {"search_query": "CVE-2017-18349 Fastjson GitHub
Pippo fix"}
```

Figure 8. Result of web scraping tool targeting vulnerability specific NIST NVD website

As seen on the picture above, the response text ends with next determined action which is using the second tool for searching the web. Observing the following calls, it can be seen that the tool based on Google Search API called Serper [71] is indeed executed with the specified query. Step after tool execution confirms that it was used successfully as the response contains a GitHub issue number #466 pointing to a correct issue and a detailed description of the currently processed vulnerability with a code of CVE-2017-18349. This was confirmed manually by using a search engine with the same following query: *CVE-2017-18349 Fastjson GitHub Pippo fix*. This returned a GitHub issue link with the same issue number as a first result. After that a summary of the mentioned CVE is constructed which includes thorough explanation of the vulnerability, affected components along with CVSSv3 impact score that were extracted from the web and recommendation for fixing it. This conclusion then gets used later in the chain for actually generating and implementing a fix for the security flaw in code.

Previously described chain of events confirms that the provided tools are used correctly and purposefully, at least when using the full-sized model. They are integrated into the flow as the author planned from the beginning - agents should use these tools to gather as much information about the vulnerability as possible before starting to fix it. Also, the results of the multi-agent system utilizing the original DeepSeek R1 [61] model show significant difference between complete and relevant fixes compared to the iteration done without tools. As the difference seems a little too big to be accidental, this confirms the usefulness of providing agents different tools they might need to achieve their goals.

6.3 Results From Previous Studies and Comparison

At the time of writing this paper, up to author's knowledge, there are no publicly available papers on the multi-agent approach on fixing vulnerable code. Despite this, there does exist research on simply prompting large language models and using singular agents for doing that. Going more specific to being able to compare this paper's results directly to other papers, author found and picked out three studies that focus on repairing vulnerable code using LLM-s and use the same dataset, Vul4J [31], as this paper. General introductions of the analyzable studies are in the literature review section. Therefore, this section focuses purely on the results of the papers and their similarities with the current paper.

A study [6] based on the GPT-4 [30] large language model used a very similar evaluation process as the current paper - manual code review paired with Vul4J dataset [31] tests. The process of extracting vulnerable code and passing it to the large language model is also similar, but not exactly same. Instead of passing the whole file to the LLM, it extracts the

source code of the vulnerable method and passes the model only this small snippet of code. Although the OpenAI models' specifications are not public and therefore the exact size of the GPT-4 model is not known, it is still categorized as a big model like the main one in the current study, DeepSeek R1 [61]. Also, according to the comparative study assessing both model's capabilities in code generation [75] they are very similar. Although taking into account the LLM input difference, the results of this paper are still comparable to those of the study based on the GPT-4 model in terms of evaluating multi-agent and traditional approach for automated vulnerability repairing. Similarly to this paper, the targeted study also targeted only single-file fixes from the dataset which left it 46 vulnerabilities to work with. The study conducted three runs that resulted in 12, 14 and 11 passing vulnerability fixes respectively. Considering the number of targeted vulnerabilities, percentages of fixed security flaws were 32.43%, 37.84% and 29.73% which results in the average of 33.33%. Additionally to evaluating purely the amount of fixed vulnerabilities, this study also evaluated the correctness of returned textual instructions. The study concluded that there is a strong correlation between correct repair instructions and fixed source code and vice versa. Although there were some exceptions where guidelines were correct, but the code could not be fixed, most of the repairing results matched with instructional results. Study admits that the paper's results are much worse compared to ones conducted on synthetic vulnerability datasets, but higher than other similar studies using the Vul4J dataset. It concluded that large language models are improving continuously and the point in time where they can be applied into real practical processes is not far. [6]

The next study that proposed their own neural network model VulMaster [32] also uses the same evaluation method as the current and also lastly analyzed paper - manual code verification and Vul4J tests. Additionally, it also targets only single-hunk vulnerabilities which makes the number of targeted vulnerabilities in this study 35. It used multiple large language models with different natures like frozen LLM-s, fine-tuned LLM-s and also APR models. Out of all their selected models, VulMaster gave the best result with repairing 9 out of 35 vulnerabilities, resulting in a repair rate of 25.7%. This was followed by Codex and InCoder models resulting in 6 successful repairs and CodeGen fixing 5 security flaws. [32]

A study using coding focused LLM-s [33] utilizes a different evaluation approach. Instead of manual review and tests, it uses a specific formula to calculate the accuracy of the generated solution. The paper doesn't explicitly mention the number of used vulnerabilities from the Vul4J [31] dataset. Judging by the highest mentioned vulnerability identifier, Vul4J-74, it is expected that the dataset is used in its original size that's covered with tests which is 79 instances. Three fine-tuned models, CodeT5, PLBART and InCoder produced 6 correct patches while CodeGen produced 7 correct patches for Vul4J dataset. Digging

deeper, the study found that providing LLM vulnerability type as a hint also resulted in 7 successfully repaired vulnerabilities while explicitly specifying vulnerable code lines resulted in 8 correct patches in Vul4J dataset scope. [33]

In conclusion, none of the three previously analyzed studies, even when LLM-s were given only the vulnerable function, resulted in a too high fixing percentage. While the 37.84% vulnerability fixing rate achieved in study using GPT-4 [6] is optimistic, it is much lower than, for example the rate of bug fixing using synthetic dataset. Although a software bug and software vulnerability are not identical concepts, they are similar enough to be compared in the context of automatic code repairing. A study focused on bug fixing [76] that also used GPT-4 large language model [30] achieved a success rate of 77.5% by using follow-up requests and providing more information on the bug. This was done on synthetic bug fixing dataset QuixBugs [77]. This again confirms that the real-world cases, whether bugs or vulnerabilities, are more complex and much more difficult to handle.

To assess the effectiveness of the used multi-agent LLM system in automated vulnerability repair, its results are compared to previous studies using the same dataset. Initial study analyzed that is based on GPT-4 model [6] reached an average fixing rate of 33.33%. Current study's best result achieved by using multi-agent system was 11 successfully repaired vulnerabilities out of 51 which translates roughly to 21.57%. Additionally, another 13 flaws were provided solutions that were very similar in terms of logic and implementation. It is very important to note that the considered paper forwarded large language model only the code of the vulnerable function rather than passing in whole file content. Considering that, results achieved by utilizing multi-agent approach are quite promising. Second study [32] managed to achieve a result of fixing 9 out of 35 security flaws using their own model called VulMaster. This result exceeds the current paper's completely fixed vulnerabilities' rate only with 4.13% with rest of the relevant solutions being excluded. Since the current study uses only one pre-trained model as it is compared to VulMaster using a combination of two LLM-s with one being fine-tuned, their solution's slightly better performance is understandable. Lastly analyzed study [33] managed to fix 10 Vul4J [31] flaws with utilizing sensitively fine-tuned model and multiple-hint prompts. Assuming that the dataset was used in its full size with 79 instances, 10 repaired vulnerabilities translates to a repair rate of 12.66%. Considering the usage of file-level approach and other studies' results, the multi-agent method for automated vulnerability repair justifies itself by resulting a fix rate of complete solutions similar or a little smaller than other studies received with additional considerable amount of other relevant fixes.

6.4 Ablation Study

In addition to the experiments conducted for main results, an ablation study was done. Ablation refers to removing one or more parts of the system to observe how it affects the behavior of the whole system [78]. In the scope of this study, the usable system consists of a base pre-trained large language model and multiple autonomous agents. Therefore, it was analyzed how removing different agents affects the vulnerability repairing performance of the multi-agent system. Since the setup used in this study relies on sequential process structure meaning that the agents are placed in a sequence and the execution order is always the same. This also means that agents are directly connected to their tasks and goals. Therefore, some of the agents were more important and could not be excluded because otherwise the system would not understand its end goal - fixing the vulnerable code.

For the first ablation experiment, a researcher agent was removed. Researcher agent's goal was to gather as much and as detailed information about the identified vulnerability. Its output was specified to include a detailed summary about the vulnerability as well as information on how to fix it in code. As a result, the 4-agent setup without the researcher managed to completely repair 8 vulnerabilities and provide relevant solutions for 7 vulnerabilities that includes 4 highly relevant and 3 minimally relevant fixes.

In the second run, additionally to removing a researcher agent, a security analyst agent was also removed. Security analyst agent's main task was to locate and describe the vulnerability in the code and provide tips for repairing it for software developer agent. Its output was configured to consist of exactly these two parts - description of the found vulnerability and recommendations for the agent that is going to repair it. Running the system without these two agents resulted in completely repairing 8 vulnerabilities and generating useful fixes for also 8 vulnerabilities from which 5 were highly relevant and 3 minimally relevant. Results for both ablation experiments are displayed in Table 4 below.

Agent setup / Evaluation Categories	Complete	Highly Relevant	Minimally Relevant	Failed
Without researcher	8	4	3	36
Without researcher and analyst	8	5	3	35
5-agent setup w/o tools & hint	5	7	6	33
5-agent setup with tools & hint	11	7	6	27

Table 4. Evaluation of different approaches.

As seen from the table above, both of the setups resulted in the same exact amount of completely repaired vulnerabilities. The number of relevant solutions is also very similar differing only by one. Although the original 5-agent setup that was provided a hint and could use tools outperforms both of the ablation experiments, their results are comparable with experiment done without providing tools and hint and experiment without agents. When the iterations done without tools and hint and without agents resulted in 5 and 6 complete results respectively then both ablation study runs outperformed them with 8 complete results. However, the configurations with three and four agents managed to provide relevant solutions for 8 and 7 vulnerabilities respectively while the setup without tools and hint generated 13 relevant solutions and setup without agents 12 relevant solutions. This means that the total number of failed solutions using reduced amount of agents is 36 and 35 respectively whereas the 5-agent setup without tools and hint resulted in 33 failures. While the original 5-agent setup with tools and hint clearly outperforms other configurations by resulting in only 27 failed solutions, the overall value of the rest of the outputs provided by using different setups is similar. As one might expect, some of the vulnerabilities are easier for LLM to repair than others. Consequently, over half of the completely fixed vulnerabilities overlapped in both iteration results that are identified by following ID-s: Vul4J-45, Vul4J-46, Vul4J-61, Vul4J-64, Vul4J-77.

Two of the last agents in sequence in original setup were developed with a focus on improving the generated result syntactically. These agents were called file cleaner and file completion agents with having a software architect and software developer backgrounds respectively. As said, these agents were not explicitly initialized to repair the vulnerability in code, but rather to improve the chance of the generated result being syntactically correct and compilable. Since neither of them were critical in the process of fixing the security flaw, an additional iteration without these agents was done. Because of the reason that they didn't directly participate in vulnerability fixing process, comparing the number of

correctly fixed security flaws was not considered meaningful. Instead, the number of directly compilable generated results was compared. Using the original 5-agent setup generated 30 solutions that could be directly replaced into the origin project and compiled successfully. By removing the file cleaner and file completion agents this result dropped to 7 within project directly compilable results. There is a significant difference of 23 instances which confirms the effectiveness of adding these two additional agents to enhance the quality of the final output.

In conclusion, removing researcher and security analyst agents resulted in a worse result than using the original 5-agent setup. This confirms the usefulness of adding independent agents to the flow that gather information about the vulnerability before starting fixing it. However, the difference between removing only the researcher and removing the researcher with security analyst is not big. Running the configuration without either of them actually resulted in one relevant solution more than running it only without the researcher. Considering that the difference in results is so minor, this indicates that their individual contributions may overlap and it's possible that one agent combining both of their goals and tasks may be enough to provide similar value. An additional experiment done without file cleaner and file completion agents whose goals was to improve the end result syntactically and make it compliant with its origin project provided a lot fewer directly compilable solutions than the original setup. Although they are not directly connected to repairing the vulnerability, they proved their importance for making the potentially vulnerability-free generated result fit into its origin project and modifying it to only include correct Java code.

6.5 Additional Observations

In addition to evaluating the accuracy and effectiveness of vulnerability repairs, several secondary behaviors and performance characteristics of the system were observed. These include unnecessary code formatting and refactoring by the LLM as well as its challenge to consistently output compilable code that's structure would align with original file's. Additionally, it was compared how much time on average it takes for the system to provide solutions for the vulnerabilities.

6.5.1 Code Formatting and Generating Compilable Files

Alongside fixing security flaws, LLM-s very often refactored and formatted other parts of code which in most cases made the code more readable, but wasn't necessary from the perspective of code repairing. This was often expressed through rewriting ternary conditional operator containing lines into full-size if-blocks or vice versa. Also, it was observed that the LLM preferred to keep long variable definitions and function calls in one line even if it was split into multiple lines in the original file. Another commonly modified element was the set of Spring [79] annotations. Although majority of the time the modifiable annotation was descriptive and not mandatory, there were cases where the LLM removed annotations that suppressed warnings or were required. Removing annotations can fail the compilation even when the rest of code might be correct and vulnerability-free. One big challenge for LLM-s were to return the complete compilable Java file that would straight fit into the origin project. Even though it was precisely described that the end result should contain all the non-vulnerable methods as the original file and its inheritance and other important properties should not be modified unless really necessary, it still struggled in some cases. Since the model was given the whole Java file without information where exactly the vulnerability in the code is located, it is, in some cases, expected. As mentioned in the ablation study, adding two additional agents to ensure the file's syntactic correctness and the presence of needed methods improved the results significantly.

6.5.2 Time Consumption

Additionally, two extra runs were conducted using different setups with intention to track the time it takes for the system to provide solutions to vulnerabilities. The first experiment utilized the multi-agent system equipped with tools while the second run used an approach without agents. Comparing the times for both of these configurations indicate if and how much does the time consumption differ between a system using multiple autonomous agents and the one without agents for repairing vulnerable code. To not use too much

resources, but still use a sufficient amount of vulnerabilities to calculate the average time, a total of 40 vulnerabilities were used. Times were measured two different ways - a time for each vulnerability from the start of the loop until the end of the loop and a total time from starting the program until the program termination. To clarify, in code, repairable vulnerabilities are handled one by one in a loop meaning that the system executes the same block of code for each vulnerability. Later, all of the independent vulnerability fixing times were summarized which allowed to calculate the average time for the system to provide a solution. The summarized time for providing solutions for all 40 instances using the multi-agent setup was 4 hours 35 minutes and 17 seconds while the whole process in total took 5 hours and 2 minutes. This means that about 27 minutes were spent on secondary activities like connecting to a Docker [43] container, writing metrics and data to files and switching between vulnerabilities. Taking the summarized time, it results in an average solution providing time of 413 seconds converting to 6 minutes and 53 seconds. Using the no-agent approach, the cumulative time taken to provide solutions for all 40 vulnerabilities was 2 hours 48 minutes and 59 seconds. Dividing this with the number of used vulnerabilities results in an average fixing time of 254 seconds. The overall time of the process was 2 hours and 59 minutes which leaves about 10 minutes for processes not directly associated with vulnerability fixing. The difference between the fastest and slowest time between the two different setups was similar. The setup without agents had a fastest fixing time of 46 seconds and slowest fixing time of 651 seconds while the multi-agent setup's times were 203 seconds and 828 seconds respectively. In conclusion, the average time for providing a solution for a vulnerability is longer for the configuration with multiple agents which is expected. Difference between average times between two setups were 159 seconds which is not too significant considering the multi-agent setup contained 5 agents. The time spent in addition to fixing the vulnerabilities didn't differ too much, but was still longer for the configuration using agents with making 8.9% of the total time while the no-agent setup spent 5.6% of its total process time for it.

7. Conclusion

As the software becomes increasingly important with it being integrated into more and more domains, its reliability and trustworthiness also rises significantly. One of the main ways to ensure software security is to keep the code free of vulnerabilities. Traditional methods for repairing and detecting vulnerabilities in code tend to lack scalability and adaptability, especially when working with complex codebases. To handle sanitizing code that changes quick and differentiates across various technologies and domains, new scalable code repairing solutions have to be developed. The rapid development and public popularization of artificial intelligence technologies fit as a suitable candidate for solving this problem.

Artificial intelligence based solutions have experienced a significant increase in their usage across various domains. Fields of software development and cybersecurity are no exceptions by already including AI based code completion and vulnerability management tools. There exist a lot of different ways of using AI in software management processes. This thesis investigated the potential of using a multi-agent system built on large language model for automated vulnerability repair on real-world Java vulnerabilities. By constructing a sequence of agents with separate goals and backgrounds like software developer, security analyst and researcher, a real development team was being simulated. Performance of the system was evaluated against a dataset containing real-world Java vulnerabilities and the results were compared to other AI utilizing studies also focusing on vulnerable code repair and using the same dataset.

Results justified the usage of multiple autonomous agents by providing similar results compared to other AI based security flawed code fixing studies. A run with best performing configuration managed to correctly fix 11 vulnerabilities with same or very similar implementation as human patch. To pass all of the dataset's tests, some of those required minor modifications like adding additional import. In addition, the same setup produced a total of another 13 relevant solutions. Of those, 7 implemented analogous logic as patch being marked as highly relevant and 6 targeting correct vulnerability without providing clear solution being categorized as minimally relevant. Also, the system showed a great performance of analyzing long Java files and implementing a vulnerability specific fix in a correct location. This lays a promising base for further experimentation with combining autonomous agents and also for handling full-sized files in the process of vulnerability repair rather than providing only the vulnerable function.

In conclusion, this thesis provides evidence that using multi-agent system is a promising direction in the field of vulnerable code repair. While it is not ready to fully replace human expertise in this area, it has a potential of significantly speeding up the process of handling security flaws originating from code thus reducing costs and minimizing the vulnerable time window. Future work in this area could focus on experimenting using the multi-agent approach based on other, potentially fine-tuned, large language models. Also, since there are a lot of various ways how to construct and combine autonomous agent with their tools and goals, it would be useful to experiment with and compare the results of using different multi-agent setups.

7.1 Limitations

During the experimentation and research, several limitations were addressed that influenced the selection of the study components and the interpretation of the results. These limitations highlight the challenges that can be addressed right in the beginning when planning to conduct similar experiments.

Analyzing the whole file

Using the approach of passing the whole source code of the vulnerable file to the large language model system prevents it from processing large files due to input size limitations. In the context of this paper, the prompt size limitation influenced only few test instances, but this can be a bigger obstacle when working with larger files.

Single evaluable experiment run for each setup

Due to resource and time limitations, although multiple experiments with different configurations were conducted, evaluation ready runs were only executed once. Since large language models are non-deterministic, running each experiment for example three times, would bring more certainty and credibility to the end results.

Using a single large language model

In the scope of this study, only one large language model was used to generate evaluable results - DeepSeek R1 [61] in two different sizes. This means that the results' credibility is heavily influenced by agent and their tools usage capability of a singular LLM. Therefore, it would be useful to experiment with the multi-agent system using other large language models like GPT-4o [30].

References

- [1] IBM. *Cost of a Data Breach Report 2024*. 2024. URL: <https://www.ibm.com/reports/data-breach>.
- [2] Sentinel One. *Key Cyber Security Statistics for 2025*. 2024. URL: <https://www.sentinelone.com/cybersecurity-101/cybersecurity/cybersecurity-statistics/>.
- [3] Quang-Cuong Bui et al. *APR4Vul: an empirical study of automatic program repair techniques on real-world Java vulnerabilities*. Article. 2023. URL: <https://link.springer.com/article/10.1007/s10664-023-10415-7>.
- [4] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. *RepairAgent: An Autonomous, LLM-Based Agent for Program Repair*. 2024. URL: <https://arxiv.org/pdf/2403.17134>.
- [5] Hammond Pearce et al. *Examining Zero-Shot Vulnerability Repair with Large Language Models*. 2023 IEEE Symposium on Security and Privacy (SP). 2023. URL: <https://ieeexplore.ieee.org/abstract/document/10179324>.
- [6] Zoltán Ságodi et al. *Reality Check: Assessing GPT-4 in Fixing Real-World Software Vulnerabilities*. 2024. URL: <https://dl.acm.org/doi/pdf/10.1145/3661167.3661207>.
- [7] Avishree Khare et al. *Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities*. 2023. URL: <https://arxiv.org/pdf/2311.16169>.
- [8] Raphaël Khoury et al. *How Secure is Code Generated by ChatGPT?* 2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC). 2023. URL: <https://ieeexplore.ieee.org/abstract/document/10394237>.
- [9] Fabio Duarte. *Number of ChatGPT Users (Apr 2024)*. Article. 2024. URL: <https://explodingtopics.com/blog/chatgpt-users>.
- [10] Global App Testing Staff Writers. *How much does software testing cost in 2024?* Article. 2024. URL: <https://www.globalapptesting.com/blog/software-testing-cost>.
- [11] Caiming Zhang and Yang Lu. *Study on artificial intelligence: The state of the art and future prospects*. Journal of Industrial Information Integration. 2021. URL: <https://doi.org/10.1016/j.jii.2021.100224>.

- [12] Frank F. Xu et al. *A systematic evaluation of large language models of code*. MAPS 2022: Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming. 2022. URL: <https://doi.org/10.1145/3520312.3534862>.
- [13] Heewoo Jun, Qiming Yuan, and Yichen Xu. *HumanEval: Hand-Written Evaluation Set*. GitHub repository. 2021. URL: <https://github.com/openai/human-eval>.
- [14] Swimm Team. *Code Completion in the Age of Generative AI*. Article. 2024. URL: <https://swimm.io/ai-tools-for-developers/code-completion-in-the-age-of-generative-ai>.
- [15] Yujia Li et al. *Competition-level code generation with AlphaCode*. 2022. URL: <https://www.science.org/doi/full/10.1126/science.abq1158>.
- [16] Relevance AI. *What is a Multi Agent System?* 2025. URL: <https://relevanceai.com/learn/what-is-a-multi-agent-system>.
- [17] *What Are Multi-Agent Systems?* 2025. URL: <https://www.nvidia.com/en-eu/glossary/multi-agent-systems/>.
- [18] Qineng Wang et al. *Rethinking the Bounds of LLM Reasoning: Are Multi-Agent Discussions the Key?* 2024. URL: <https://arxiv.org/pdf/2402.18272>.
- [19] Simeng Han et al. *FOLIO: Natural Language Reasoning with First-Order Logic*. 2022. URL: <https://arxiv.org/abs/2209.00840>.
- [20] Taicheng Guo et al. *Large Language Model based Multi-Agents: A Survey of Progress and Challenges*. 2024. URL: <https://arxiv.org/pdf/2402.01680>.
- [21] IBM. *What is an intrusion detection system (IDS)?* 2023. URL: <https://www.ibm.com/topics/intrusion-detection-system>.
- [22] Quamar Niyaz, Weiqing Sun, and Ahmad Y Javaid. *A Deep Learning Based DDoS Detection System in Software-Defined Networking (SDN)*. 2016. URL: <https://arxiv.org/pdf/1611.07400.pdf>.
- [23] Prateek Dewan and Ponnurangam Kumaraguru. *Towards automatic real time identification of malicious posts on Facebook*. 2015 13th Annual Conference on Privacy, Security and Trust (PST). 2015. URL: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7232958>.
- [24] OWASP. *Source Code Analysis Tools*. 2024. URL: https://owasp.org/www-community/Source_Code_Analysis_Tools.

- [25] Roland Croft et al. *An Empirical Study of Rule-Based and Learning-Based Approaches for Static Application Security Testing*. 2021. URL: <https://dl.acm.org/doi/pdf/10.1145/3475716.3475781>.
- [26] Saikat Chakraborty et al. *Deep Learning based Vulnerability Detection: Are We There Yet?* 2020. URL: <https://arxiv.org/pdf/2009.07235>.
- [27] Zhen Li et al. *VulDeePecker: A Deep Learning-Based System for Vulnerability Detection*. 2018. URL: <https://arxiv.org/pdf/1801.01681>.
- [28] Xin Zhou, Ting Zhang, and David Lo. *Large Language Model for Vulnerability Detection: Emerging Results and Future Directions*. 2024. URL: <https://dl.acm.org/doi/pdf/10.1145/3639476.3639762>.
- [29] Quanjun Zhang et al. *Pre-Trained Model-Based Automated Software Vulnerability Repair: How Far are We?* IEEE Transactions on Dependable and Secure Computing. 2023. URL: <https://ieeexplore.ieee.org/abstract/document/10232867>.
- [30] OpenAI. *GPT-4*. 2025. URL: <https://openai.com/index/gpt-4>.
- [31] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. *Vul4J: A Dataset of Reproducible Java Vulnerabilities Geared Towards the Study of Program Repair Techniques*. 2022. DOI: 10.1145/3524842.3528482.
- [32] Xin Zhou et al. *Multi-LLM Collaboration + Data-Centric Innovation = 2x Better Vulnerability Repair*. 2024. URL: <https://arxiv.org/pdf/2401.15459>.
- [33] Guodong Zhang et al. *Automated Program Vulnerability Repair Based on Fine-Grained Cue Optimization and Local Sensitive Fine-Tuning*. 2025. URL: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5090066.
- [34] CrewAI. *CrewAI - The leading open source platform*. 2024. URL: <https://www.crewai.com/open-source>.
- [35] Yizheng Chen et al. *DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection*. 2023. URL: <https://surrealyz.github.io/files/pubs/raid23-diversevul.pdf>.
- [36] Jiahao Fan et al. *A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries*. 2020. URL: <https://ieeexplore.ieee.org/document/10148758>.
- [37] Ollama. *Ollama*. 2024. URL: <https://ollama.com/search>.
- [38] Ollama. *llama3*. 2024. URL: <https://ollama.com/library/llama3>.
- [39] *Bearer CLI*. 2025. URL: <https://github.com/Bearer/bearer>.

- [40] *OWASP Top Ten*. 2025. URL: <https://owasp.org/www-project-top-ten/>.
- [41] *CWE Top 25 Most Dangerous Software Weaknesses*. 2025. URL: <https://cwe.mitre.org/top25/index.html>.
- [42] *Bearer - Configure the scan to meet your needs*. 2025. URL: <https://docs.bearer.com/guides/configure-scan/>.
- [43] *Docker*. 2025. URL: <https://www.docker.com/>.
- [44] *Vul4J Docker image*. 2024. URL: <https://hub.docker.com/r/bquongas/vul4j>.
- [45] *Use containers to Build, Share and Run your applications*. 2025. URL: <https://www.docker.com/resources/what-container/>.
- [46] *AgentOps. Trace, Debug, Deploy Reliable AI Agents*. 2025. URL: <https://www.agentops.ai>.
- [47] *Frameworks To Build Multi-Agent AI Applications*. 2024. URL: <https://getstream.io/blog/multiagent-ai-frameworks/>.
- [48] *AutoGen*. 2025. URL: <https://microsoft.github.io/autogen/stable/>.
- [49] *AutoGen - AgentChat*. 2025. URL: <https://microsoft.github.io/autogen/stable/user-guide/agentchat-user-guide/index.html>.
- [50] *Runtime Logging with AutoGen*. 2025. URL: https://microsoft.github.io/autogen/0.2/docs/notebooks/agentchat_logging/.
- [51] Vanna Winland, Meredith Syed, and Anna Gutowska. *What is crewAI*. 2024. URL: <https://www.ibm.com/think/topics/crew-ai>.
- [52] *LangChain*. 2025. URL: <https://www.langchain.com/>.
- [53] *CrewAI. CrewAI Tools*. 2025. URL: <https://docs.crewai.com/concepts/tools>.
- [54] *CrewAI. CrewAI Introduction*. 2025. URL: <https://docs.crewai.com/introduction>.
- [55] *CrewAI. CrewAI Quickstart*. 2025. URL: <https://docs.crewai.com/quickstart>.
- [56] *LangGraph*. 2025. URL: <https://langchain-ai.github.io/langgraph/>.
- [57] *LangGraph Quickstart*. 2025. URL: <https://langchain-ai.github.io/langgraph/tutorials/introduction/>.

- [58] *LangGraph - How to stream*. 2025. URL: <https://langchain-ai.github.io/langgraph/how-tos/streaming/>.
- [59] Guru Bhandari, Amara Naseer, and Leon Moonen. *CVEfixes: automated collection of vulnerabilities and their fixes from open-source software*. 2021. URL: <https://dl.acm.org/doi/10.1145/3475960.3475985>.
- [60] National Institute of Standards and Technology. *National Vulnerability Database*. 2024. URL: <https://nvd.nist.gov/>.
- [61] *DeepSeek*. 2025. URL: <https://www.deepseek.com/>.
- [62] *What is Azure AI Foundry*. 2025. URL: <https://learn.microsoft.com/en-us/azure/ai-foundry/what-is-azure-ai-foundry>.
- [63] *DeepSeek-R1 Release*. 2025. URL: <https://api-docs.deepseek.com/news/news250120>.
- [64] *Introducing OpenAI o1*. 2025. URL: <https://openai.com/o1/>.
- [65] *DeepSeek vs. OpenAI: Comparing the New AI Titans*. 2025. URL: <https://www.datacamp.com/blog/deepseek-vs-openai>.
- [66] *DeepSeek - Models Pricing*. 2025. URL: https://api-docs.deepseek.com/quick_start/pricing.
- [67] *API Pricing*. 2025. URL: <https://openai.com/api/pricing/>.
- [68] *TalTech AI-lab Resources*. 2025. URL: <https://ai-lab.pages.taltee/en/intro/>.
- [69] *DeepSeek-R1 671B: Complete Hardware Requirements*. 2025. URL: <https://dev.to/askyt/deepseek-r1-671b-complete-hardware-requirements-optimal-deployment-setup-2e48>.
- [70] *CrewAI. Tools - Google Serper Search*. 2025. URL: <https://docs.crewai.com/tools/serperdevtool>.
- [71] *Serper. The World's Fastest Cheapest Google Search API*. 2025. URL: <https://serper.dev>.
- [72] *CrewAI. Tools - Scrape Website*. 2025. URL: <https://docs.crewai.com/tools/scrapewebsitetool>.
- [73] *Common Weakness Enumeration - A Community-developed List of SW HW Weaknesses That Can Become Vulnerabilities*. 2025. URL: <https://cwe.mitre.org/index.html>.
- [74] *AnythingLLM. Why Is My Agent Not Using Tools*. 2025. URL: <https://docs.anythingllm.com/agent-not-using-tools>.

- [75] Md Motaleb Hossen Manik. *ChatGPT vs. DeepSeek: A Comparative Study on AI-Based Code Generation*. 2025. URL: <https://arxiv.org/pdf/2502.18467>.
- [76] Dominik Sobania et al. *An Analysis of the Automatic Bug Fixing Performance of ChatGPT*. 2023. URL: <https://ieeexplore.ieee.org/document/10189263>.
- [77] *QuixBugs Benchmark*. 2017. URL: <https://github.com/jkoppel/QuixBugs>.
- [78] *Machine Learning: What Is Ablation Study?* 2025. URL: <https://www.baeldung.com/cs/ml-ablation-study>.
- [79] *Spring Framework*. 2025. URL: <https://spring.io/projects/spring-framework>.

Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis¹

I Martin Kilgi

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Evaluating the Effectiveness of Multi-Agent Large Language Model System for Automated Vulnerable Code Repair”, supervised by Hayretdin Bahsi
 - 1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
 - 1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons’ intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

18.05.2025

¹The non-exclusive licence is not valid during the validity of access restriction indicated in the student’s application for restriction on access to the graduation thesis that has been signed by the school’s dean, except in case of the university’s right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 - Software Developer Agent and Its Task

```
software_developer_agent = Agent(  
    role = "Senior_Software_Developer",  
    goal = """Analyze found Java software vulnerability and  
        implement a fix for it.  
        Keep the leftover code present and untouched""",  
    backstory = """  
        You're a seasoned Java software developer that also has  
        a background of  
        cybersecurity who's task is to analyze the software  
        vulnerability and develop a fix for it.""",  
    verbose = True,  
    llm = llm,  
    tools=[serper_dev_tool]  
)  
  
software_developer_task = Task(  
    agent = software_developer_agent,  
    description = """  
        Fix the found vulnerability in Java code and implement  
        in into existing code.  
        CVE code of the vulnerability is {cve_id}.  
        - Take into account the recommendations and information  
        that others have gathered about the vulnerability.  
        - Do not change Java class' constructor or inheritance  
        properties.  
        - Do not delete or rewrite any existing code, only  
        replace or refactor the vulnerable part.  
        - The Java class name has to be {file_name} and it has  
        to extend or implement exact same classes as the  
        original code.  
        Here is the vulnerable code: {vulnerable_java_code}  
        """,  
    expected_output = """  
        Repaired vulnerability-free version of the inputted Java  
        file that compiles without errors.  
        """,  
)
```


Appendix 3 – Security Analyst Agent and Its Task

```
security_analyst_agent = Agent(  
    role = "Senior_Security_Analyst",  
    goal = """  
        Find a vulnerability in the provided Java code and  
        provide a thorough description of the vulnerability  
        that helps the software developer develop and  
        implement a fix for it."""  
    backstory = """  
        You're a security analyst whose task is to review  
        vulnerable Java code and find a vulnerability from it  
        .  
        Since you have extensive knowledge in cybersecurity, you  
        can provide a lot useful information about the  
        vulnerable code for repairing it."""  
    verbose = True,  
    llm = llm,  
    tools=[bearer_sast_tool]  
)  
  
security_analyst_task = Task(  
    agent = security_analyst_agent,  
    description = """  
        Following code snippet contains vulnerability with  
        Common Vulnerabilities and Exposures (CVE) ID of {  
        cve_id}.  
        Find it in the code and provide a detailed description  
        of the vulnerability and recommendations for fixing  
        it.  
        Here is the vulnerable code: {vulnerable_java_code}""",  
    expected_output = """  
        Thorough description of the found vulnerability in the  
        code and recommendations for the software developer  
        that is  
        going to repair it.  
        """  
)
```

Appendix 4 – Research Agent and Its Task

```
research_agent = Agent(  
    role = "Cybersecurity_Researcher",  
    goal = """  
        Find information and details about the vulnerability.  
    """,  
    backstory = """  
        You are an experienced cybersecurity researcher with a  
        valuable skill of finding extensive information about  
        vulnerabilities and providing detailed information  
        about them.  
    """,  
    llm = llm,  
    verbose = True,  
    tools = [web_tool, serper_dev_tool]  
)  
  
research_agent_task = Task(  
    agent = research_agent,  
    description = """  
        Find as detailed and useful information as possible  
        about the vulnerability that would help in fixing it.  
        Common Vulnerabilities and Exposures ID of the  
        vulnerability is {cve_id}.  
    """,  
    expected_output = """  
        A detailed summary of the vulnerability and information  
        on how to repair it in the code.  
    """,  
)
```

Appendix 5 – File Cleaning Agent and Its Task

```
code_sanitization_agent = Agent(  
    role = "Software_Architect",  
    goal = """  
        Remove all the additional information and explanations  
        from the result. Return only pure Java code.  
        Ensure that the code compiles and is ready for  
        production.  
    """,  
    backstory = """  
        You are a software architect with long experience  
        specialized in Java.  
    """)  
  
code_sanitization_task = Task(  
    agent = code_sanitization_agent,  
    description = """  
        Remove all the explanations, comments and other  
        additional information from the output and return  
        only pure Java code file.  
        Java class name must be {file_name}.  
    """,  
    expected_output = """  
        Clear and pure Java code that compiles without errors.  
    """)
```

Appendix 6 – File Completion Agent and Its Task

```
code_integrity_agent = Agent(  
    role = "Software_developer",  
    goal = """  
        Compare Java code generated by other agent that you get  
        as an input with original vulnerable code.  
        Make sure that the generated Java code contains rest of  
        the methods from original code that were not  
        associated with vulnerability repairing.  
        Keep in mind that the fixed Java file must not contain  
        entrypoint method as it is part of existing project.  
        """,  
    backstory = """  
        You are a software agent with long experience that is  
        specialized in Java programming language.  
        """,  
)  
  
code_integrity_task = Task(  
    agent = code_integrity_agent ,  
    description = """  
        Add all methods that were not associated with the  
        vulnerability from the original Java code to the  
        generated end result Java code.  
        Goal of this is to ensure that the generated Java file  
        fits into existing project perfectly and compiles  
        without problems.  
        Original Java code is following: {vulnerable_java_code}  
        """,  
    expected_output = """  
        Fixed vulnerability-free Java code that is as similar to  
        the official vulnerable Java code containing all  
        methods that are not vulnerable.  
        """,  
)
```

Appendix 7 – System and User Message of A Single Prompt

```
SystemMessage( """  
    You are a senior Java developer with cybersecurity  
    background.  
    Your goal is to fix the provided Java code without modifying  
    the code that is not related to the vulnerability.  
    """)
```

```
UserMessage(f """  
    Fix the following vulnerable Java code without modifying  
    unnecessary parts of it.  
    Since the Java file is part of a bigger project, changing  
    its structure can break the compilation process.  
    Provide the fixed Java code without vulnerabilities that  
    compiles.  
    The vulnerable code contains a vulnerability with a CVE code  
    of {cve_id}.  
    Here is the vulnerable Java code: {vulnerable_code}  
    """)
```