Leveraging Intra- and Inter-References in Vulnerability Detection Using Multi-Agent Collaboration Based on LLMs

Chung-Nan Tsai[1], Jingnan Xie[2], Chun-Ming Lai[3] and Ching-Sheng Lin[4]

1 Lam Research Japan GK, Kanagawa 222-0033, Japan; chung-nan.tsai@lamresearch.com

2 Computer Science, Millersville University of PA, Millersville, PA 17551, USA; jingnan.xie@millersville.edu

3 Department of Computer Science, Tunghai University, Taichung City 407, Taiwan; cmlai@thu.edu.tw

4 Master Program of Digital Innovation, Tunghai University, Taichung City 40704, Taiwan; cslin612@thu.edu.tw

Corresponding author: Ching-Sheng Lin

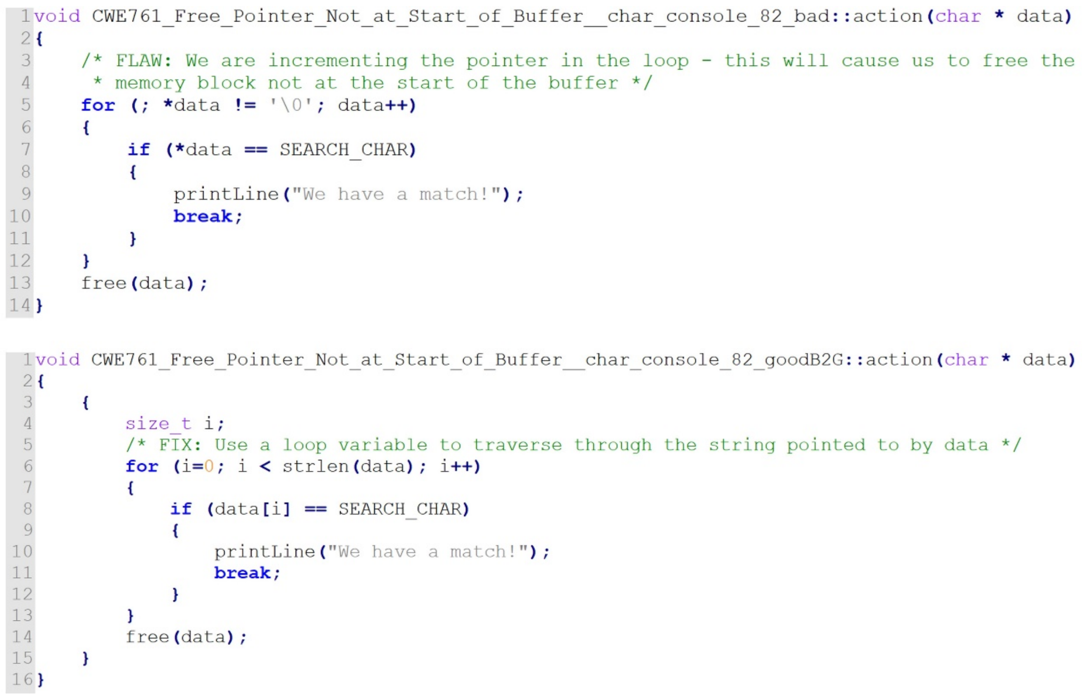
Abstract

As AI technology advances, early detection of code vulnerabilities becomes increasingly critical for preventing exploitation, reducing remediation costs, enhancing user trust, and improving system performance. Recently, Large language Models (LLMs) have shown remarkable performance across various tasks using few-shot learning. This study aims to adopt LLM-based agents in a multi-turn discussion framework enhanced with a Retrieval-Augmented Generation (RAG) strategy to improve response quality. Our multi-agent approach transforms a single LLM into a highly collaborative intelligence through multi-turn self-collaboration with diverse personas. By leveraging their combined expertise, this approach enhances the accuracy of code vulnerability detection and improves inference capabilities. To effectively utilize few-shot learning samples, we employ intra-references to extract a small number of similar samples from the training data, and inter-references to obtain samples from external data. The experimental studies are conducted on the TreeVul_Ext dataset, and our model achieves promising results across different evaluation criteria. Furthermore, we apply our model to detect code vulnerabilities in low-resource languages and exhibit competitive performance as well.

Keywords Code vulnerabilities, large language models, retrieval-augmented generation, multi-agent approach, intra-references, inter-references

1. Introduction

Software security is crucial for both individuals and businesses, but the existence of software vulnerabilities provides opportunities for attackers to exploit, potentially leading to severe consequences and damages. Vulnerabilities in enterprise systems often result in large-scale data breaches and financial losses, while vulnerabilities in consumer applications can expose sensitive information such as personal identification, account details and login credentials [1]. The majority of software projects, particularly open-source software (OSS), rely on developers to proactively identify vulnerabilities embedded in the code [2]. With the rapid increase in software complexity and the growing demand for information security knowledge, it is becoming increasingly difficult for developers to manually identify code vulnerabilities. This challenge underscores the urgent need for the development of automated software vulnerability detection. As an example from the Vuldeepecker dataset [3], consider the incorrect memory deallocation shown at the top of Figure 1, where the pointer is incremented before being freed and causes potential undefined behavior. In contrast, the correct implementation at the bottom of Figure 1 ensures proper memory deallocation by preserving the original pointer which prevents such issues. Although this is a simplified example, real-world vulnerabilities are often more complex. Nonetheless, it effectively demonstrates the importance of code security.



```c
void CWE761_Free_Pointer_Not_at_Start_of_Buffer__char_console_82_bad::action(char * data)
{
    /* FLAW: We are incrementing the pointer in the loop - this will cause us to free the
     * memory block not at the start of the buffer */
    for (; *data != '\0'; data++)
    {
        if (*data == SEARCH_CHAR)
        {
            printLine("We have a match!");
            break;
        }
    }
    free(data);
}
```

```c
void CWE761_Free_Pointer_Not_at_Start_of_Buffer__char_console_82_goodB2G::action(char * data)
{
    {
        size_t i;
        /* FIX: Use a loop variable to traverse through the string pointed to by data */
        for (i=0; i < strlen(data); i++)
        {
            if (data[i] == SEARCH_CHAR)
            {
                printLine("We have a match!");
                break;
            }
        }
        free(data);
    }
}
```

Fig. 1 A vulnerable code example (top) and its secure version (bottom).

Prior research on vulnerability detection can be generally classified into pattern-based and similarity-based methods [4-6]. Pattern-based vulnerability detection methods rely on manually defined rules by experts, making the process time-consuming and prone to errors. These methods often struggle with high false-positive and false-negative rates due to the subjective nature of rule creation [7-9]. Similarity-based vulnerability detection methods compare code fragments to known vulnerabilities using similarity metrics with the goal of identifying potential vulnerabilities caused by code cloning. While effective for detecting cloned vulnerabilities, these methods are limited in their ability to discover new vulnerabilities, resulting in a high false negative rate for vulnerabilities stemming from other factors [10-12]. Due to deep learning's (DL) capability to capture complex features, numerous researchers have shifted their focus towards utilizing DL techniques to learn source code representations for vulnerability detection. These approaches can effectively identify a wide range of vulnerabilities by learning from known examples, thereby improving generalization and contextual understanding of source code [13].

Trained on extensive and diverse datasets, large language models (LLMs) have demonstrated impressive capabilities across numerous domains [14-15]. Their broad applicability stems from the comprehensive knowledge acquired during the training process, which enables them to generalize effectively to various tasks and applications. Moreover, recent studies have shown the notable potential of multi-agent collaboration based on LLMs by simulating human-like interactions and employing the combined intelligence of multiple agents. Through iterative rounds of dialogue and cooperation, this approach significantly enhances the model's reasoning capabilities and leads to more sophisticated decision-making competence [16-17]. Although LLMs are gradually being utilized in software engineering field [18], the use of multi-agent approaches for security-oriented tasks, particularly in vulnerability detection, remains relatively underexplored [19]. Our research aims to bridge this gap by transforming a single LLM into a highly collaborative intelligence through multi-turn self-collaboration with diverse agents, leveraging their combined expertise to enhance the accuracy of code vulnerability detection and improving inference capabilities. In our multi-agent system, different agents are designed with distinct personas. Therefore, the terms "agent" and "persona" are used interchangeably in this paper.

While LLMs have shown remarkable potential in different fields, they also face inherent limitations that affect their ability to generate accurate responses [20]. Retrieval-Augmented Generation (RAG) is an advanced technique that combines retrieval models with LLMs to enhance the quality of AI-generated answers. This approach augments the generative capabilities of LLMs by retrieving relevant information from supplementary data sources. The key advantage of RAG lies in its ability to provide factual support for the generated responses [21]. In this research, we explore two types of data sources for agents to reference. The first type, which we refer to as intra-references, consists of reference training data that is used without prior training. The second type, designated as inter-references, utilizes software weakness types from external sources. Leveraging these intra- and inter-references, agents can engage in meaningful discussions and collaborate effectively to enhance their problem-solving capabilities for software vulnerability detection.

The key innovations and main contributions of our method can be outlined as follows.

- We propose a multi-agent collaboration framework based on LLMs and leverage the RAG strategy in a training-free manner. Our approach investigates how agent communication facilitates consensus decision-making in software vulnerability detection. To maximize the effectiveness of different resources through RAG, we integrate training data with QuixBugs examples [22] as few-shot learning samples to enhance the agents' decision-making capabilities.
- We assess our approach on the TreeVul_Ext dataset for software vulnerability detection [23], and show competitive performance against state-of-the-art methods. Additionally, to assess robustness, we test our model on a low-resource language dataset, where it continues to deliver impressive results [24].

We structure the remainder of this paper as follows. Section 2 reviews various research fields, technologies and paradigms relevant to our study. In Section 3, an in-depth discussion of our multi-agent model based on LLMs for vulnerability detection is provided. Section 4 presents experimental studies on different public datasets to assess the model's effectiveness and analyze the resulting outcomes. Finally, the last section concludes the paper and outlines potential avenues for future research.

2. Related works

Given the importance of vulnerability detection in software quality assurance, numerous researchers have developed methods for its automatic detection. We categorize these methods into three types—traditional detection, learning-based detection, and LLM-based detection—and discuss them in this section.

Traditional code vulnerability detection can include static analysis and rule-based methods. Static methods for vulnerability detection involve analyzing source code or binaries without executing them and aim to identify potential security issues by examining the code's structure and data flow [25-26]. Although these methods are efficient for early-stage detection, they may produce false positives due to their inability to confirm runtime behavior [27]. Rule-based techniques identify software vulnerabilities through predefined patterns or manually defined rules applied to the code and its execution [28-29]. However, the diverse nature of vulnerabilities can make predefined approaches insufficient to cover all possible types, leading to false positives or false negatives that undermine detection accuracy. Furthermore, creating these rules and patterns is a labor-intensive and error-prone task that depends heavily on manual input.

Recent advances in the software security community have led to the application of machine learning and neural network approaches, including Tree models [30], BiLSTM [3], CNN [31] and Graph-based methods [32], for detecting vulnerable source code. These models have shown promise in identifying security issues by learning patterns and relationships within the code. Besides above data-driven approaches, pre-trained models have also been used effectively in vulnerability detection. Models like CodeBERT [33] and GraphCodeBERT [34] leverage pre-training techniques such as masked language modeling and next sentence prediction to enhance code representation and improve the accuracy of security issue detection.

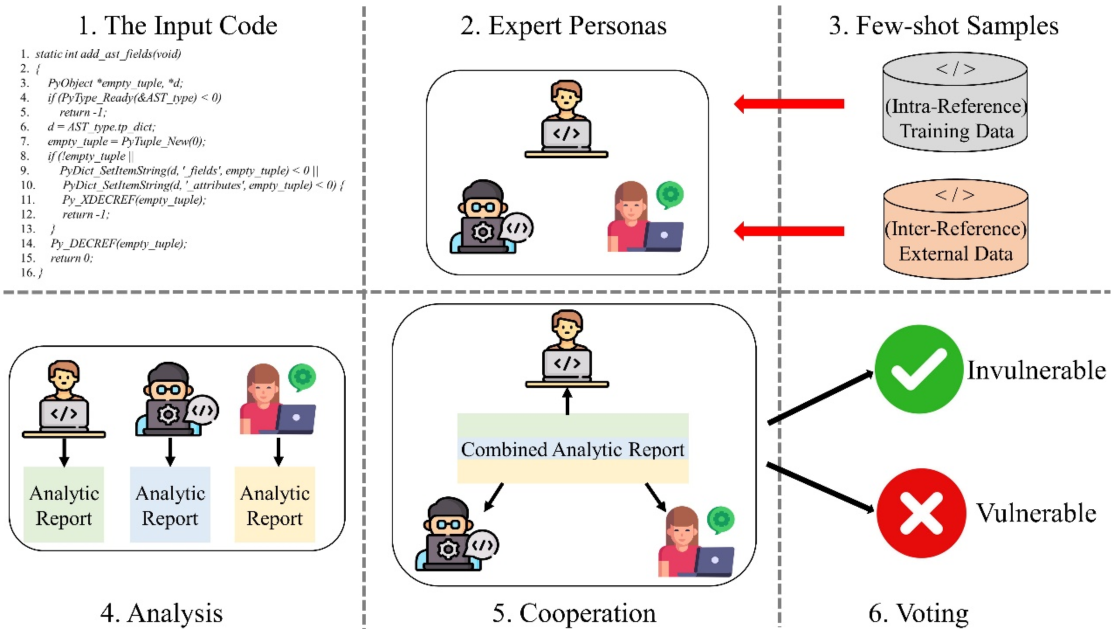
In recent years, the advancement of LLMs has significantly contributed to the growing popularity of AI. Related techniques such as RAG, fine-tuning, and prompt engineering have enabled various domains to leverage LLMs efficiently for diverse applications. The multi-role consensus-based approach simulates the roles of developers and testers throughout the software development lifecycle. It detects and classifies code vulnerabilities through collaborative discussion and consensus-building processes [35]. VulTrial is a mock-court multi-agent framework that leverages LLMs to simulate prosecutor, defense attorney, judge, and jury for collaborative vulnerability detection [36]. Different fine-tuning approaches have been explored for adapting LLMs in vulnerability detection. LLMAO only fine-tunes bidirectional adapter layers to overcome LLMs' left-to-right nature. ProveRAG is an LLM-powered system that analyzes vulnerabilities with retrieval augmentation from web data and applies a self-critique mechanism to reduce omissions and hallucinations [37]. iAudit is a prompt-based framework that fine-tunes large language models for smart contract auditing. It employs a two-stage decision-making and justification process, incorporating LLM-based agents to iteratively refine vulnerability explanations [38]. LLM-SmartAudit is a framework that enhances smart contract auditing through role specialization and action execution to ensure efficient collaboration and coherent analysis [39].

3. Methodology

In this section, we begin by defining the problem of software vulnerability detection. Following this, we present our framework, multi-agent collaboration based on LLMs with intra- and inter-references, in detail.

3.1. Problem description

Software vulnerability detection can be performed at various levels. Our focus is on function-level detection, aiming to determine whether a specific function is vulnerable according to the given source code. We define the mapping function: f : X → Y for this detection problem where X represents the source code of the function and Y = {0, 1} is the label set with 1 indicating vulnerability and 0 denoting invulnerability. This function is designed using LLM prompts which allow multiple agents to collaborate in detecting vulnerabilities in the code. By leveraging prompt-based interactions among agents, the system identifies the security status of the code and facilitates the automated identification of vulnerabilities without traditional training.



```
static int add_ast_fields(void)
{
    PyObject *empty_tuple, *d;
    if (PyType_Ready(&AST_type) < 0)
        return -1;
    d = AST_type.tp_dict;
    empty_tuple = PyTuple_New(0);
    if (!empty_tuple ||
        PyDict_SetItemString(d, '_fields', empty_tuple) < 0 ||
         PyDict_SetItemString(d, '_attributes', empty_tuple) < 0) {
         Py_XDECREF(empty_tuple);
         return -1;
     }
     Py_DECREF(empty_tuple);
     return 0;
}
```

Fig. 2 The system architecture.

3.2. Proposed framework

Our proposed framework, multi-agent collaboration based on LLMs with intra- and inter-references, is described in the Figure 2 and consists of six stages: (1) The Input Code: Source code is given to experts to review and analyze; (2) Expert Personas: LLM-based personas play the role of expert agents; (3) Few-shot Samples: Several examples are provided to guide persona performance; (4) Analysis: Expert personas perform assessments based on their expertise; (5) Cooperation: All experts' results are shared to aid their decisions.; and (6) Voting: The final result is determined by a majority vote of the experts. Our multi-agent cooperation architecture is inspired by human collective intelligence, where individuals leverage Theory of Mind (ToM)—the ability to attribute mental states to others—to enhance collaboration. Human societies excel at utilizing teamwork and specialization to achieve shared goals. By assigning distinct personas to each agent, we emulate this cognitive capability, allowing agents to interpret the intentions of their counterparts and achieve collective intelligence [40-41].

We implement a training-free methodology for vulnerability detection. There are two separate datasets: X^{test} which comprises input code samples with pending vulnerability assessment and X^{train} which contains labeled code examples with known vulnerability statuses. Rather than utilizing X^{train} for model training, we employ it exclusively to provide relevant examples to our expert personas for reference.

Our approach incorporates a module leveraging LLM personas specifically tailored for vulnerability detection. These personas are established through carefully designed prompts, which guide the model's behavior and decision-making process. We employ three distinct personas (denoted as persona(p)), each focusing on different aspects of vulnerability detection to ensure a comprehensive analysis:

- persona(1): a senior security consultant who specializes in risk assessment and compliance, leveraging industry standards such as OWASP, NIST, and CWE to evaluate vulnerabilities and recommend best practices for secure software development.
- persona(2): a seasoned software engineer who has extensive experience in secure coding practices and software architecture, focusing on identifying security flaws in code implementation and suggesting robust engineering solutions to mitigate potential threats.
- persona(3): an experienced ethical hacker who actively performs penetration testing and exploits security weaknesses, providing insights into real-world attack vectors and helping to strengthen defenses by simulating adversarial tactics.

RAG has been considered to be a game-changing technique in enhancing the capabilities of LLMs, particularly in tasks that require up-to-date and domain-specific information. To maximize the benefits of RAG for vulnerability detection, we utilize a dual-reference system by leveraging both intra-references and inter-references mechanisms. The intra-references work by identifying k code samples most similar to the code under examination (represented as x_i^{test}) from X^{train}.

$$\mathrm{Intra}_i = \left\{ (x_j^{train}, y_j^{train}) \middle| \ j \in \mathrm{TopK}(\mathrm{sim}\left(x_i^{test}, X^{train}\right)) \right\} \tag{1}$$

where sim denotes the similarity function, the TopK function returns the k indices with the highest similarity scores and the value of k is set to 5 in this research. On the other hand, the inter-references incorporate k' vulnerable code samples (denoted as Inter^{+}) and k' non-vulnerable code samples (denoted as Inter^{-}) from an external database that differs from the target programming language. In this study, k' is set to 20. It is worth noting that the intra-references mechanism employs an online and dynamic knowledge construction process, whereas the inter-references mechanism utilizes an offline knowledge base. Consequently, each test sample is associated with a unique set of intra-references samples, based on its similarity to the training samples. In contrast, the inter-references samples remain consistent across all test data, providing a fixed set of external reference information.

Given an input source code x_i^{test} with its reference samples Intra_i and $\mathrm{Inter}^{+} \odot \mathrm{Inter}^{-}$, each expert persona initially will be asked to generate an analytic report following the associated guideline prompt:

$$\mathrm{Report}_i^{Initial}(p) = \mathrm{LLM}(\mathrm{persona}(p), x_i^{test}, \mathrm{Intra}_i, \mathrm{Inter}^{+} \odot \mathrm{Inter}^{-}) \tag{2}$$

where the prompt is defined at the top of Figure 3. The expert persona, combined with direct prompting and few-shot learning techniques, can ensure that different personas contribute diverse insights and improve self-knowledge of LLMs to some extent.

In the previous stage of analysis, each persona independently performs judgment and analysis based solely on the given input and its corresponding retrieval results, without any interaction with other personas. To enhance the decision-making process, we introduce a second round of collaborative refinement. In this stage, each persona gains access to the judgments and analyses from all other personas in the previous phase, enabling mutual collaboration. This iterative process allows for more refined and informed assessments to improve the overall accuracy and robustness of vulnerability detection. The prompt for this stage is shown at the bottom of Figure 3.

$$\mathrm{Report}_i^{Final}(p) = \mathrm{LLM}(\mathrm{persona}(p), x_i^{test}, \mathrm{Report}_i^{Initial}) \tag{3}$$

Finally, we aggregate the results from the second round using a voting mechanism to determine whether the given code is vulnerable. Each persona's refined judgment from the second round contributes to the final decision, ensuring a more robust and well-rounded assessment. By incorporating a two-round evaluation process, this collaborative approach enhances detection accuracy by allowing personas to refine their analysis based on prior assessments, ultimately leveraging diverse perspectives for a more comprehensive evaluation. The final output is represented as $\hat{y}_i^{test}$ for x_i^{test}.

You are {persona(p)}
Here are some examples of C/C++ source code, each followed by an analysis of its vulnerability status:
$\{\mathrm{Intra}_i\}$
Additionally, here are some examples of python source code, each followed by an analysis of its vulnerability status:
$\{\mathrm{Inter}^+ \odot \mathrm{Inter}^-\}$
Based on these examples, analyze the following C/C++ code to determine if it is vulnerable. First, answer 'vulnerable' or 'non-vulnerable,' and then provide a brief explanation:
Code to analyze: $\{x_i^{test}\}$

You are {persona(p)}
Here are the results from Round 1 analysis:
$\{\mathrm{Report}_i^{\mathrm{Initial}}(1)$
$\mathrm{Report}_i^{\mathrm{Initial}}(2)$
$\mathrm{Report}_i^{\mathrm{Initial}}(3)\}$
Please reconsider your analysis of the following C/C++ code based on the above results. Answer 'vulnerable' or 'non-vulnerable,' and briefly explain why.
Code to analyze: $\{x_i^{test}\}$

Fig. 3 Two-round prompts used in code vulnerability assessment

The complete process of vulnerability detection is detailed in Algorithm 1. The input includes the training set, the testing set, and inter-references. The output is the prediction results of the testing set. At the beginning, personas are set up in lines 1–3. For each testing sample x_i^{test}, the first step is to obtain the intra-references, Intra_i, from the training set. Next, each persona is asked to generate an initial report based on Eq. (2) in lines 7-9. Subsequently, final reports are created according to the initial reports and Eq. (3) in lines 11-13. In the final step, a majority voting scheme is applied to obtain the final vulnerability prediction of the testing code.

Algorithm 1: Vulnerability Detection Using Multi-Agent Collaboration

Input: Training data $\{X^{train}, Y^{train}\}$, testing data $\{X^{test}, Y^{test}\}$, $\mathrm{Inter}^+ \odot \mathrm{Inter}^-$.
Output: Vulnerability detection result $\{\hat{Y}^{test}\}$.

```
1.  for p in P:
2.    setup persona(p)
3.  end for
4.  for $x_i^{test}$ in $X^{test}$:
5.    Obtain $Intra_i$ for $x_i^{test}$
6.    #Initial stage
7.    for p in P:
8.      Generate $Report_i^{Initial}(p)$ for p using Eq. (2)
9.    end for
10.   #Final stage
11.   for p in P:
12.     Given $Report_i^{Initial}$, generate $Report_i^{Final}(p)$ for p using Eq. (3)
13.   end for
14.   #Voting scheme
15.   $\hat{y}_i^{test}$ = majority-vote($Report_i^{Final}(1), \ldots Report_i^{Final}(p)$ )
16. end for
17. return $\hat{y}^{test}$
```

4. Experiments

This section presents our experimental evaluation of the proposed software vulnerability detection method. Our analysis includes five parts: (1) a detailed description of the vulnerability datasets used in our experiments; (2) the definition of performance metrics; (3) extensive comparisons against state-of-the-art vulnerability detection techniques; (4) detailed ablation studies demonstrating the key contributions of each component in our model and (5) experimental validation on an additional dataset to demonstrate model transferability. Our experiments aim to explore the following two research questions regarding software vulnerability prediction.
RQ1: How does our proposed model compare to state-of-the-art LLMs in performance?
RQ2: How do different modules within our model impact its performance in vulnerability detection?

4.1. Dataset

Our analysis is based on TreeVul_Ext [18], an extended version of the TreeVul dataset of vulnerability-fixing commits [23]. In vulnerability-fixing patches, all functions containing line modifications are labeled as vulnerable (positive samples). On the contrary, functions within the same file that contain no line modifications are considered non-vulnerable (negative samples). TreeVul utilizes a dataset comprising 1,560 repositories, while TreeVul_Ext focuses on 20 repositories specifically implemented in C/C++. TreeVul_Ext consists of 7,683 functions in the training set, 853 functions in the validation set, and 386 functions in the test set. The test set comprises 193 vulnerable functions and 193 non-vulnerable functions. In our setting, the training set serves as the data source for intra-references, while the test set is used for evaluation. For each test sample, we retrieve the five most similar samples from the training set based on similarity and use them as intra-references.

For inter-references, we utilize the QuixBugs dataset [22], focusing on the Python language. We randomly select 20 positive samples and 20 negative samples as reference results.

4.2. Evaluation metrics

The evaluation metrics for software vulnerability detection include four standard performance indicators: accuracy (Acc), precision (Pre), recall (Rec), and F1-score (F1). Accuracy measures the overall correctness of the model by evaluating the proportion of correctly classified source codes. Precision assesses the model's ability to correctly identify actual vulnerabilities, while recall indicates how well the model retrieves all vulnerable code snippets. F1-score, defined as the harmonic mean of precision and recall, provides a balanced overall assessment of the model's performance. The formulas for these four metrics are defined below:

$$\mathrm{Acc} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}} \tag{4}$$

$$\mathrm{Pre} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} \tag{5}$$

$$\mathrm{Rec} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \tag{6}$$

$$\mathrm{F1} = \frac{2 \times \mathrm{Pre} \times \mathrm{Rec}}{\mathrm{Pre} + \mathrm{Rec}} \tag{7}$$

where TP (True Positive) represents the number of correctly identified vulnerabilities, TN (True Negative) denotes the number of correctly classified non-vulnerabilities, FP (False Positive) refers to instances where non-vulnerable code is mistakenly classified as vulnerable, and FN (False Negative) indicates cases where actual vulnerabilities are not detected by the model.

4.3. Experimental performances and analysis

In this section, we will introduce several benchmark models for comparison with our proposed method where most of them are LLM-based approaches widely applied in vulnerability prediction.

- Zero-shot prompting [44]: Zero-shot prompting with GPT-3.5 is used as a baseline for code vulnerability prediction, leveraging pre-trained knowledge without additional fine-tuning.
- GPT-3.5 [44]: ChatGPT, developed by OpenAI, is a popular and powerful language model. It has demonstrated exceptional performance across various tasks [45], making it a widely used tool for natural language processing applications, and security-related research is no exception. GPT-3.5-Turbo API is used to generate responses.
- Llama-3-8B [46]: Llama-3-8B is a publicly available LLM developed by Meta, trained on 15 trillion tokens with an improved tokenizer supporting a 128K vocabulary [47]. It features 8 billion parameters and an 8,192-token context window, enhancing its efficiency in language modeling tasks.
- Gemma-7B [48]: Gemma-7B is a lightweight LLM developed by Google, featuring 7 billion parameters and a context window of 8,192 tokens. It is trained using a combination of supervised fine-tuning (SFT) on text-only synthetic and human-generated data, along with reinforcement learning from human feedback (RLHF), which significantly improves its performance in downstream evaluations and human preference assessments.
- Mixtral-8x7B [49]: Developed by Mistral AI, Mixtral-8x7B is an innovative large language model that utilizes a Mixtures of Experts (MoE) architecture, featuring eight distinct models with 7 billion parameters each and a router network that dynamically selects two expert models to process individual tokens. Pre-trained on open web data, the model offers a substantial context length of 32,768 tokens, making it particularly powerful for handling complex instruction-based tasks with nuanced and efficient language processing.
- GPT-4o [50]: GPT-4o is a multimodal LLM which is capable of processing text, speech, and vision, and enables advanced applications such as image recognition and text generation in more than 50 languages [51]. Its processing speed is double that of GPT-4 Turbo, marking a significant advancement in human-machine interaction and digital communication.

- Comb. Voting: This is a technique for aggregating results from above five LLMs using majority voting. It enhances output reliability by comparing and selecting the most consistent response across different model predictions.
- CodeBERT [33]: CodeBERT is a transformer-based model pre-trained on a large corpus of source code and natural language, making it highly effective for various code-related tasks. It has been recognized as a state-of-the-art model for software vulnerability prediction [42-43]. Its strong performance is largely attributed to the availability of high-quality vulnerable training datasets.
- GraphCodeBERT [34]: This approach utilizes Transformer architecture and incorporates two structure-aware pre-training tasks, edge prediction and node alignment. These tasks enable the model to effectively capture code semantics by leveraging data flow relationships.
- Multi-role consensus-based approach (MCBA) [35]: This method simulates developer-tester interactions through a maximum of three discussion rounds, terminating early if consensus on vulnerability identification is achieved.
- iAudit [38]: The original iAudit framework combines fine-tuned models with LLM-based agents to enable explainable smart contract auditing. We simulate four reasoning roles (Detector, Reasoner, Ranker, and Critic) purely through prompt-based interaction without any fine-tuning.
- VulTrial [36]: We employ prompt-based simulation to emulate the roles and interactions of the security researcher, code author, moderator, and review board as defined in the VulTrial framework.

All experimental studies are performed on a Windows-based system with the following specifications: an Intel Core i9-11900K (3.50GHz) processor, 128GB of system memory, and an NVIDIA GeForce RTX 3090 with 24GB of dedicated memory. The implementation of our model utilizes GPT-3.5-turbo for the LLM.

The comparison results are presented in Table 1, where the methods are divided into four categories: zero-shot prompting, a series of few-shot prompting (ranging from GPT-3.5 to Comb. Voting) [52], training methods (CodeBERT and GraphCodeBERT), and the multi-agent approaches (MCBA, iAudit and VulTrial). The zero-shot prompting method serves as the baseline, with the lowest F1 score of 31.3%. The few-shot prompting methods utilize prompt techniques and provide two types of code examples. The first set consists of randomly selected code examples from the training data, while the second set includes the most semantically similar code examples. Our method outperforms the best single model, GPT-4o, with a 6.8% improvement in F1-score. Additionally, we surpass the Comb. Voting model by 14.2% in F1-score. Compared to the trainable models, our F1-score shows an improvement of 26.5% over CodeBERT and 12.4% over GraphCodeBERT. In comparison to multi-agent models, our model achieves an F1-score improvement of 17.7%, 13.8% and 29.9% over methods MCBA, iAudit and VulTrial, respectively. As observed, our model achieves the highest accuracy, recall, and F1-score among all compared models, demonstrating its effectiveness. Our persona-based multi-agent design prioritizes recall by encouraging diverse and cautious reasoning patterns, which helps capture a wide range of potential vulnerabilities. This approach aligns with the safety-first principle in vulnerability detection, where missing a real issue is more critical than raising false alarms. As a result, while precision may decrease, the high recall makes our method suitable for early-stage detection and large-scale analysis scenarios. Moreover, we conduct McNemar's test to compare our method with the second-best performing approach (GPT-4o). The test yields a p-value of 0.0446, indicating a statistically significant difference at the 0.05 level. In summary, the results and analysis presented above effectively address our RQ1.

In addition to the discussion on our method's better performance than others, we have also observed several noteworthy findings. First, even single-language models like Gemma-7B and GPT-4o can achieve comparable or even better performance than CodeBERT and GraphCodeBERT. This suggests that in-context learning with examples allows non-finetuned LLMs to outperform trained models, highlighting the effectiveness of prompt-based approaches for vulnerability detection. Second, except for the zero-shot prompting method, all other LLM-based models outperform GPT-3.5. However, GPT-3.5 exhibits the highest precision among all methods while having the lowest recall. This suggests that GPT-3.5 is more conservative in making predictions, favoring correctness over completeness, which may result in missing many vulnerable cases and increasing security risks in enterprises. Third, although combining the results of all LLMs using a voting-based approach achieves good accuracy (68.4%), its F1-score is lower than that of single-language models like Gemma-7B and

GPT-4o. Among the models we compared, GPT-4o performs particularly well—not only ranking second in F1-score but also striking a good balance between recall and precision.

While our method shows potential for improvement, a limitation of our method is its computational cost, which can be analyzed in terms of time. The zero-shot prompting baseline achieves an average inference time of 0.47 seconds, while GPT-3.5 with few-shot prompting averages 1.23 seconds. The most time-efficient model, GraphCodeBERT, requires only 0.01 seconds on average. In contrast, MCBA, which involves multi-turn discussions between two agents, takes an average of 9.57 seconds. Our approach has an average inference time of 4.66 seconds, primarily due to the need for six OpenAI API calls. To mitigate this limitation, refining the prompt structure to make it more concise could reduce processing times while maintaining response quality.

Table 1 Experimental results of different models (%).

Model	Acc	Pre	Rec	F1
Zero-shot prompting	48.9	47.8	23.3	31.3
GPT-3.5	62.7	**76.3**	36.8	49.7
LLaMA-3-8B	60.9	63.5	51.3	56.7
Gemma-7B	67.6	71.8	58.0	64.2
Mixtral-8x7B	63.2	73.4	41.5	53.0
GPT-4o	67.4	66.8	68.9	67.9
Comb. Voting	68.4	75.2	54.9	63.5
CodeBERT	60.3	62.3	53.3	57.3
GraphCodeBERT	59.0	56.9	74.6	64.5
MCBA	57.7	56.4	67.8	61.6
iAudit	59.3	57.5	71.5	63.7
VulTrial	55.2	55.1	56.5	55.8
Our Model	**69.7**	66.4	**79.8**	**72.5**

4.4. Ablation studies

To evaluate the fusion effectiveness of our proposed model, we conduct ablation studies by breaking down the model into four variants:

- w/o 2nd round: The model makes vulnerability decisions through a voting mechanism after performing just one round of analysis.
- w/o references: This model employs a two-round analysis framework to determine vulnerability, without incorporating any reference material.
- w/o intra-references: The model determines vulnerability through a two-round analysis framework, incorporating only Python references from inter-references rather than C/C++ references from intra-references.
- w/o inter-references: This model applies a two-round analysis framework to determine vulnerability, utilizing only C/C++ references from intra-references while excluding Python references from inter-references.

Table 2 illustrates the results of the experimental comparison from the ablation study in terms of accuracy, precision, recall and F1-score. Our model achieves the best performance among all variations in accuracy, recall, and F1-score. The second-best performance is observed in the intra-references model, which achieves an F1-score of 68.7, significantly outperforming the inter-references model (F1-score 58.9). This suggests that leveraging training data as reference samples provides greater benefits than relying on external data sources with different programming languages. Furthermore, the model performs the worst when the second-round collaboration mechanism is not employed (F1-score 39.6), highlighting the effectiveness of multi-agent collaboration in improving model performance. Our ablation study demonstrates that each module contributes to performance improvement and provides insights into RQ2.

Table 2 Results from the ablation studies (%).

Model	Acc	Pre	Rec	F1
Our Model	**69.7**	66.4	**79.8**	**72.5**
w/o 2nd round	58.0	**70.1**	27.5	39.6
w/o references	63.7	64.9	59.6	62.2
w/o intra-references	58.1	57.7	60.1	58.9
w/o inter-references	64.8	61.8	77.2	68.7

4.5. Application to the low-resource language

In addition to the TreeVul_Ext dataset, we expand our experiments to include emerging programming languages to further validate the effectiveness of our proposed model for vulnerability detection. This task is commonly referred to as low-resource software vulnerability prediction. In this paper, we select Kotlin as the target, an emerging programming language that serves as a compelling alternative to Java and is rapidly gaining popularity among Android software developers [53]. The dataset is from prior work [24], and contains 20 vulnerable functions and 98 non-vulnerable functions. A 10-round evaluation is conducted for the models to ensure robustness and mitigate performance fluctuations caused by the limited dataset size. In each round, the vulnerable and non-vulnerable functions are sampled and divided into 60% for training, 20% for validation and 20% for testing, respectively. We use precision, recall and F1-score to evaluate the model performance. GPT Fine-tuning, GPT Few-shot and CodeBERT are state-of-the-art models chosen to evaluate the effectiveness of our proposed method. The experimental results are shown in Table 3. Our model achieves the best results in terms of F1-score. Similar to the findings on the C/C++ dataset, our recall performance remains outstanding. This encouraging performance gives us confidence in extending the model to explore various aspects of software vulnerability research.

Table 3 Performance evaluation on the Kotlin dataset.

Model	Pre	Rec	F1
GPT Fine-Tuning	0.37	0.32	0.34
GPT Few-Shot	**0.43**	0.44	0.43
CodeBERT	0.24	0.47	0.32
Our Model	0.33	**0.85**	**0.48**

5. Conclusion and future work

In this paper, we focus on software vulnerability detection and propose an LLM-based multi-agent approach that utilizes multi-turn cooperation with intra- and inter-references strategies. Each agent is assigned a specific persona and conducts analysis by referencing codes from the training data that are similar to the test samples, as well as external codes in different languages for vulnerability reference. In addition to independent analysis, agents collaborate by sharing their findings to enhance the overall evaluation process. Our proposed model is validated on the C/C++ based TreeVul_Ext dataset and has demonstrated remarkable performance. Moreover, we broaden our investigation by evaluating the model on the Kotlin dataset, a low-resource language dataset, where it demonstrates equally impressive performance.

Despite achieving promising results on both the primary dataset and the extended dataset, our approach still has areas for further improvement. First, while current model demonstrates high recall rates, there is still room for improvement in precision. This imbalanced outcome suggests the need for more sophisticated filtering mechanisms to reduce false positives while maintaining the high detection rate of vulnerabilities. Second, this paper currently focuses solely on binary detection, and future work could enhance the system by identifying the precise vulnerability localization within the code. Finally, the current time consumption is relatively high due to the multiple API calls to OpenAI's model with longer inputs. To address this, future work could explore the use of prompt refinement techniques to optimize the inference process and enable faster analysis.

Data Availability

Not applicable.

Ethical Approval

Not applicable.

Conflict of interest

The authors declare that they have no competing interests.

References

1. Grahn, D., Chen, L., & Zhang, J. (2024). Vul-Mixer: Efficient and Effective Machine Learning–Assisted Software Vulnerability Detection. Electronics, 13(13), 2538.
2. Tian, Z., Tian, B., Lv, J., Chen, Y., & Chen, L. (2024). Enhancing vulnerability detection via AST decomposition and neural sub-tree encoding. Expert Systems with Applications, 238, 121865.
3. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., ... & Zhong, Y. (2018). Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681.
4. Zhang, J., Liu, Z., Hu, X., Xia, X., & Li, S. (2023). Vulnerability detection by learning from syntax-based execution paths of code. IEEE Transactions on Software Engineering, 49(8), 4196-4212.
5. Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., & Chen, Z. (2021). Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing, 19(4), 2244-2258.
6. Nguyen, S., Nguyen, T. T., Vu, T. T., Do, T. D., Ngo, K. T., & Vo, H. D. (2024). Code-centric learning-based just-in-time vulnerability detection. Journal of Systems and Software, 214, 112014.
7. Sui, Y., & Xue, J. (2018). Value-flow-based demand-driven pointer analysis for C and C++. IEEE Transactions on Software Engineering, 46(8), 812-835.
8. Ma, X., Yan, J., Wang, W., Yan, J., Zhang, J., & Qiu, Z. (2021, November). Detecting memory-related bugs by tracking heap memory management of C++ smart pointers. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 880-891). IEEE.
9. Duan, X., Wu, J., Du, M., Luo, T., Yang, M., & Wu, Y. (2021, October). MultiCode: A Unified Code Analysis Framework based on Multi-type and Multi-granularity Semantic Learning. In 2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) (pp. 359-364). IEEE.
10. Li, J., & Ernst, M. D. (2012, June). CBCD: Cloned buggy code detector. In 2012 34th International Conference on Software Engineering (ICSE) (pp. 310-320). IEEE.
11. Kim, S., Woo, S., Lee, H., & Oh, H. (2017, May). Vuddy: A scalable approach for vulnerable code clone discovery. In 2017 IEEE symposium on security and privacy (SP) (pp. 595-614). IEEE.
12. Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., ... & McConley, M. (2018, December). Automated vulnerability detection in source code using deep representation learning. In 2018 17th IEEE international conference on machine learning and applications (ICMLA) (pp. 757-762). IEEE.
13. Hin, D., Kan, A., Chen, H., & Babar, M. A. (2022, May). LineVD: statement-level vulnerability detection using graph neural networks. In Proceedings of the 19th international conference on mining software repositories (pp. 596-607).
14. Park, J. S., O'Brien, J., Cai, C. J., Morris, M. R., Liang, P., & Bernstein, M. S. (2023, October). Generative agents: Interactive simulacra of human behavior. In Proceedings of the 36th annual acm symposium on user interface software and technology (pp. 1-22).
15. Weyssow, M., Zhou, X., Kim, K., Lo, D., & Sahraoui, H. (2023). Exploring parameter-efficient fine-tuning techniques for code generation with large language models. arXiv preprint arXiv:2308.10462.
16. Xi, Z., Chen, W., Guo, X., He, W., Ding, Y., Hong, B., ... & Gui, T. (2023). The rise and potential of large language model based agents: A survey. arXiv preprint arXiv:2309.07864.

17.Wang, Z. M., Peng, Z., Que, H., Liu, J., Zhou, W., Wu, Y., ... & Peng, J. (2023). Rolellm: Benchmarking, eliciting, and enhancing role-playing abilities of large language models. arXiv preprint arXiv:2310.00746.
18.Zhou, X., Zhang, T., & Lo, D. (2024, April). Large language model for vulnerability detection: Emerging results and future directions. In Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results (pp. 47-51).
19.Lee, C., Xia, C. S., Huang, J. T., Zhu, Z., Zhang, L., & Lyu, M. R. (2024). A Unified Debugging Approach via LLM-Based Multi-Agent Synergy. arXiv preprint arXiv:2404.17153.
20.Jeong, M., Sohn, J., Sung, M., & Kang, J. (2024). Improving medical reasoning through retrieval and self-reflection with retrieval-augmented large language models. arXiv preprint arXiv:2401.15269.
21.Ding, Y., Fan, W., Ning, L., Wang, S., Li, H., Yin, D., ... & Li, Q. (2024). A survey on rag meets llms: Towards retrieval-augmented large language models. arXiv preprint arXiv:2405.06211.
22.Lin, D., Koppel, J., Chen, A., & Solar-Lezama, A. (2017, October). QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity (pp. 55-56).
23.Pan, S., Bao, L., Xia, X., Lo, D., & Li, S. (2023, May). Fine-grained commit-level vulnerability type prediction by CWE tree structure. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) (pp. 957-969). IEEE.
24.Le, T. H. M., Babar, M. A., & Thai, T. H. (2024, June). Software Vulnerability Prediction in Low-Resource Languages: An Empirical Study of CodeBERT and ChatGPT. In Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (pp. 679-685).
25.Awadhutkar, P., Santhanam, G. R., Holland, B., & Kothari, S. (2019, August). DISCOVER: detecting algorithmic complexity vulnerabilities. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 1129-1133).
26.Lipp, S., Banescu, S., & Pretschner, A. (2022, July). An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis (pp. 544-555).
27.Charoenwet, W., Thongtanunam, P., Pham, V. T., & Treude, C. (2024, September). An empirical study of static analysis tools for secure code review. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 691-703).
28.Nguyen, T. D., Pham, L. H., Sun, J., Lin, Y., & Minh, Q. T. (2020, June). sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (pp. 778-788).
29.Tsankov, P., Dan, A., & Drachsler-Cohen, D. (2022). Practical security analysis of smart contracts. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (pp. 67-82).
30.Aivatoglou, G., Anastasiadis, M., Spanos, G., Voulgaridis, A., Votis, K., & Tzovaras, D. (2021, July). A tree-based machine learning methodology to automatically classify software vulnerabilities. In 2021 IEEE International Conference on Cyber Security and Resilience (CSR) (pp. 312-317). IEEE.
31.Qiang, G. (2022). Research on software vulnerability detection method based on improved CNN model. Scientific Programming, 2022(1), 4442374.
32.Chakraborty, S., Krishna, R., Ding, Y., & Ray, B. (2021). Deep learning based vulnerability detection: Are we there yet?. IEEE Transactions on Software Engineering, 48(9), 3280-3296.
33.Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.
34.Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., ... & Zhou, M. (2020). Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366.
35.Mao, Z., Li, J., Jin, D., Li, M., & Tei, K. (2024, July). Multi-role consensus through llms discussions for vulnerability detection. In 2024 IEEE 24th International Conference on Software Quality, Reliability, and Security Companion (QRS-C) (pp. 1318-1319). IEEE.
36.Widyasari, R., Weyssow, M., Irsan, I. C., Ang, H. W., Liauw, F., Ouh, E. L., ... & Lo, D. (2025). Let the Trial Begin: A Mock-Court Approach to Vulnerability Detection using LLM-Based Agents. arXiv preprint arXiv:2505.10961.
37.Fayyazi, R., Trueba, S. H., Zuzak, M., & Yang, S. J. (2024). ProveRAG: Provenance-Driven Vulnerability Analysis with Automated Retrieval-Augmented LLMs. arXiv preprint arXiv:2410.17406.
38.Ma, W., Wu, D., Sun, Y., Wang, T., Liu, S., Zhang, J., ... & Liu, Y. (2024). Combining fine-tuning and llm-based agents for intuitive smart contract auditing with justifications. arXiv preprint arXiv:2403.16073.
39.Wei, Z., Sun, J., Zhang, Z., Zhang, X., Li, M., & Hou, Z. (2024). LLM-SmartAudit: Advanced Smart Contract Vulnerability Detection. arXiv preprint arXiv:2410.09381.
40.Li, H., Chong, Y. Q., Stepputtis, S., Campbell, J., Hughes, D., Lewis, M., & Sycara, K. (2023). Theory of mind for multi-agent collaboration via large language models. arXiv preprint arXiv:2310.10701.
41.Tran, K. T., Dao, D., Nguyen, M. D., Pham, Q. V., O'Sullivan, B., & Nguyen, H. D. (2025). Multi-Agent Collaboration Mechanisms: A Survey of LLMs. arXiv preprint arXiv:2501.06322.

42. Fu, M., & Tantithamthavorn, C. (2022, May). Linevul: A transformer-based line-level vulnerability prediction. In Proceedings of the 19th International Conference on Mining Software Repositories (pp. 608-620).
43. Steenhoek, B., Rahman, M. M., Jiles, R., & Le, W. (2023, May). An empirical study of deep learning models for vulnerability detection. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) (pp. 2237-2248). IEEE.
44. https://platform.openai.com/docs/models/overview, [Accessed 02-06-2025]
45. Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., & Zhang, J. M. (2023, May). Large language models for software engineering: Survey and open problems. In 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE) (pp. 31-53). IEEE.
46. https://ai.meta.com/blog/meta-llama-3/, [Accessed 02-06-2025]
47. Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., ... & Ganapathy, R. (2024). The llama 3 herd of models. arXiv preprint arXiv:2407.21783.
48. Team, G., Mesnard, T., Hardin, C., Dadashi, R., Bhupatiraju, S., Pathak, S., ... & Kenealy, K. (2024). Gemma: Open models based on gemini research and technology. arXiv preprint arXiv:2403.08295.
49. Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., ... & Sayed, W. E. (2024). Mixtral of experts. arXiv preprint arXiv:2401.04088.
50. https://openai.com/index/hellogpt-4o/, [Accessed 02-06-2025]
51. Shahriar, S., Lund, B. D., Mannuru, N. R., Arshad, M. A., Hayawi, K., Bevara, R. V. K., ... & Batool, L. (2024). Putting gpt-4o to the sword: A comprehensive evaluation of language, vision, speech, and multimodal proficiency. Applied Sciences, 14(17), 7782.
52. Widyasari, R., Lo, D., & Liao, L. (2024). Beyond ChatGPT: Enhancing Software Quality Assurance Tasks with Diverse LLMs and Validation Techniques. arXiv preprint arXiv:2409.01001.
53. Ardito, L., Coppola, R., Malnati, G., & Torchiano, M. (2020). Effectiveness of Kotlin vs. Java in android app development tasks. Information and Software Technology, 127, 106374.