TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Matvei Mõskiv 242765IVCM

Ethereum Smart Contract Vulnerability Detection Using Multi-Agent Claude Model

Master's thesis

Supervisor: Hayretdin Bahsi
Research Professor

Tallinn 2026

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Matvei Mõskiv 242765IVCM

Ethereumi nutilepingute haavatavuste tuvastamine Claude'i-põhise mitmeagendi süsteemi abil

Magistritöö

Juhendaja: Hayretdin Bahsi
Research Professor

Tallinn 2026

Authors'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: Matvei Mõskiv

16.04.2026

Abstract

As of 2026 Ethereum blockchain is holding more than $56 [1] billion total value locked in smart contracts. Because smart contracts are immutable and involve monetary transactions, adversaries are motivated to exploit vulnerabilities for financial gain. While existing research has explored static analysis tools and Large Language Models for vulnerability detection, multi-agent architectures using multiple specialised instances with distinct roles and tools remain underexplored.

This work evaluates a role-based multi-agent framework, orchestrated via n8n, consisting of three agents: a Smart Contract Developer relying on general knowledge, a Smart Contract Security Specialist augmented with Slither reports, and a Smart Contract Auditor utilising SolidityScan reports. A majority voting consensus mechanism was applied, with zero-shot prompting and no fine-tuning. The framework was evaluated on the publicly available SmartBugs Curated dataset of 143 vulnerable and 30 benign smart contracts sourced from OpenZeppelin, and compared against Slither and a standalone baseline Claude model under both forced and selective tool usage configurations.

The multi-agent framework with forced tool usage achieved the highest macro-average Precision, Recall and F1-score of 65.6% and Accuracy 56.8%, outperforming Slither and the baseline model. Selective tool usage analysis revealed that agents performed better without tool invocation, suggesting that tool quality directly impacts agentic reasoning. The findings suggest that a multi-agent approach is a viable starting point when computational resources or training data are limited.

List of abbreviations and terms

LLM - Large Language Model

TVL - Total value locked

DAO - Decentralized Autonomous Organization

AI - Artificial Intelligence

GPT - Generative Pre-trained Transformer

LLaMA - Large Language Model Meta AI

API - Application Programming Interface

URL - Uniform Resource Locator

CFG - Control Flow Graph

EVM - Ethereum virtual machine

AST - Abstract Syntax Tree

CVE - Common Vulnerabilities and Exposures

RAG - Retrieval Augmented Generation

JSON - JavaScript Object Notation

Table of contents

1 Introduction

“Smart contracts are computer programs that can be executed by a network of mutually distrusting nodes, without the need of external third party authority” [2]. They are typically written in Solidity language and are executed upon interaction. Due to growth of such technology, at the time of writing, Ethereum blockchain alone has total value locked (TVL) surpassing $56 billion as of 2026 [1]. Because of the fact that smart contracts are immutable and involve monetary transactions, using cryptocurrency, the interest of adversaries in such technology also grows. “For instance, the DAO (Decentralized Autonomous Organization) hack of 2016 resulted in the loss of millions of dollars due to a vulnerability in a smart contract” [3] and throughout the year 2025, over 3.4 billion [4] were lost due to different attacks, exploits and contract vulnerabilities.

To mitigate financial losses associated with vulnerabilities introduced in the smart contracts, recent research has increasingly focused on using Large Language Models (LLM) for vulnerability detection, with several studies such as [5]-[7] demonstrating strong performance. However, despite these advances, multi-agent architectures using multiple instances of LLM with specialised roles and tools remain underexplored in the existing literature. Furthermore, as LLM evolves, more powerful models are introduced. One such model is Claude, an Anthropic's [8] advanced LLM, which has not been extensively evaluated for smart contract vulnerability detection despite its capabilities.

This work addresses this gap by evaluating role-based multi-agent architecture built on Claude 4.5 Sonnet for Ethereum smart contract vulnerability detection and comparing it against static analysis tool and standalone baseline model.

1.1 Main objective

The potential of combining the multi-agent approach based on the Claude LLM version 4.5 Sonnet, presents an opportunity to improve vulnerability detection in smart contracts. The multi-agent orchestration was built by utilizing 3rd party n8n [9] service, a tool for automation workflow that uses AI capabilities for business process automation. The system built upon the n8n service was using the role-based specialisation per agent with distinct rules to follow. Majority voting approach and zero-shot prompt technique were applied, which allowed to mitigate results that would end up being a tie between the agents. Such a system did not intend to implement fine-tuning techniques, this was done to demonstrate practical applicability when computational resources are limited. However, a majority voting consensus was implemented between specialised agents, which provided insights into LLM collaboration for security analysis. Finally the performance of the multi-agent Claude system was compared against Slither, an established static analysis tool for Solidity smart contracts, and a baseline Claude model, using standardised performance metrics.

This work aimed to expand beyond GTP (Generative Pre-trained Transformer) and LLaMA (Large Language Model Meta AI) models and include Claude's latest, at the time of writing, model 4.5 Sonnet in the evaluation of vulnerability detection of Ethereum smart contracts. The key point is to demonstrate the capabilities of the Claude multi-agent system with a role-based zero-shot approach, which is a possible alternative for fine-tuning approaches. And potentially establish a benchmark of performance metrics for future Claude models in the research aimed to expand in this field with a similar setup. In addition this work looked into the performance of tool-augmented approach, where the agents were provided with the tools, but in two separate scenarios. The first scenario, "Multi-agent with forced tool usage performance" looked into the performance of agents where tools are provided as part of the process and agents use information as is. The second scenario, "Multi-agent with selective tool usage performance" looked into performance of agents where the tools were provided, but the agents had a choice to use the information provided for the final output, during the decision-making process.

1.2 Research questions

The primary question of this work was to understand “How effective is a multi-agent framework compared to traditional static analysis tools for smart contract vulnerability detection?“.
The research was additionally guided by sub-questions such as “How effectively does a role-based multi-agent framework detect vulnerabilities across standard evaluation metrics, including recall, precision, and F1-score?”. The need for such a question arises from the limitations of deterministic analysis tools that often lack contextual reasoning required to detect logic-based vulnerabilities. By evaluating performance metrics, the research will show if a multi-agent approach provides a viable and balanced solution for vulnerability detection tasks. Moreover, during the research the attention will be on the limitations of the workflow and the approach. As limitations will point to the flaws of the system, it is a good indication of future research improvement. This question intends to explore the potential reason or the limitations behind missing or misclassifying a particular class of vulnerability during the output from a multi-agent system. And “How does forced tool usage compare to selective tool usage in terms of agentic performance?”. With such a question the research aims to understand how forced and selective tool usage affects the performance and reliability of vulnerability detection in an agentic system.

In a forced tool usage setup, the agent is provided with a report as part of the input which makes it mandatory to incorporate into each decision-making process. The selective tool usage aligns with general agentic behavior, where tools are provided as additional mechanisms that allow the agent to determine if and when to invoke the tool based on its internal reasoning.

To the best of the author’s knowledge, such comparison remains underexplored in the existing literature, particularly in the context of tool-augmented multi-agent frameworks.
The work intends to answer these questions on the basis of comparing results to existing tools and a Claude baseline model.

1.3 Research contribution

The research done in this work tried to expand the opportunities related to the vulnerability detection using Claude LLM, as knowledge gained would be used to increase the security of the blockchain ecosystem and would allow cybersecurity analysts to utilize the insights for the future work. Additionally, this work showed how convenient and applicable the setup can be for everyday analysts. In addition, the setup showed how the multi-agent framework can be utilized to protect investor's capital, by locating vulnerabilities that may exist in the smart contract of a project. The work calculated and compared the performance of a multi-agent framework against traditional static analysis tools and baseline. High performance leads to more accurate results, which could be the deciding factor whether to invest into the project if vulnerability existence is applicable to the project. This work is also beneficial for the projects themselves, as the application of the LLM for vulnerability detection adds an additional layer of audit and control, with minimum expenses.

1.4 Scope and goal

This research will utilize a multi-agent framework based on the Claude model. The workflow will incorporate a parallel decision-making process where input is processed by the agents independently. The final result of the vulnerability class is presented by the consensus. Such an approach leverages the diverse perspective of the agents and potentially increases the depth of analysis and potentially removes hallucination. The scope focuses on the zero-shot capabilities of the agents without the use of fine-tuning or few-shot examples, which allow to evaluate the performance of reasoning power in multi-agentic architecture.

This approach allowed testing agents on the selected toolset, prompt requirements and Claude model's general knowledge that it was trained by the Anthropic company. Additional goal was to evaluate force tool usage against selective tool usage by the agents. Claude's model was selected as it remains underexplored in the context of smart contract vulnerability detection despite its capabilities, making its evaluation a meaningful contribution to this field. Specifically, Claude version 4.5 Sonnet was selected as it was the newest version at the time of writing and

researching this topic. The voting on if the vulnerability existed in the smart contracts, was done by the agents. In case there would be a disagreement between the agents a new round of discussions was introduced. The iteration of discussions would take place a maximum of three times to finalize agents' mutual decisions. The majority of voting decided the outcome, meaning, if two out of three agents decided on a particular class of vulnerability - that class would be considered as the finalised result. For the data SmartBugs [10] and OpenZeppelin [11] datasets were used, which focuses on Solidity programming language. No additional external, combined or non publicly available datasets were used in this research. Such a decision was made to ensure future reproducibility of the experiment, to minimize potential points of bias, compromise in the results and to ensure transparency.

1.5 Chapter overview

This work begins with an introduction, outlining the main objective, research questions and the research contribution, followed by the scope of the research. A literature review of the relevant to the field topic is then presented, followed by a description of the research gap analysis and methodology. The results and limitations are then discussed and presented. The thesis continues with discussions where areas for future research are identified and ends with conclusions.

2 Literature Review

This chapter reviews existing work on smart contract vulnerability detection, covering static analysis tools, available datasets, Large Language Model approaches, and multi-agent frameworks, in order to identify the research gap this work addresses.

2.1 Static analysis tools for Ethereum smart contracts

To mitigate financial losses associated with vulnerabilities, a lot of effort was put into researching vulnerability detection methods. Some of the earliest tools were Oyente [12], Mythril [13], Slither [14] and Manticore [15]. These tools became widely spread due to being open to the public as the tools for vulnerability detection.

Oyente, a symbolic execution tool exclusively designed to analyze Ethereum smart contracts, which follows the execution model of Ethereum smart contracts and directly works with Ethereum Virtual Machine (EVM) byte code without access to the high level representation [16]. Oyente is open source and is available for public use from the project page [12].

For the limitations Oyente takes approximately 350 seconds to process one smart contract, mishandled exceptions that the tool flags require manual intervention and validation as such issues have to be distinguished between the real vulnerabilities and false positives [16].

Mythril is another analysis tool that uses the bytecode of the smart contract to decompile it back into EVM opcode that operates based on instructions [13], it is compatible with Ethereum, Hedera, Vechain, Tron, Quorom, Roostock and any EVM compatible blockchains [17]. However, it too has limitations.

The vulnerability detection for Mythril is only successful in 27% cases of known vulnerabilities in the annotated dataset, the rest are left undetected [18]. By design, Mythril cannot detect Bad Randomness, a vulnerability that occurs when predictable variables are used for random number generation and Short Addresses that allows malformed addresses to be passed. In addition Mythril has a high SNR (signal-to-noise ratio). In this regard SNR is the proportion of the total amount of warnings to confirmed vulnerabilities. High SNR indicates a larger number of

warnings compared to actual vulnerabilities. As an example, the study by [18] reported 215 vulnerability warnings, while only 31 were identified as actual tagged vulnerabilities. And as all static analysis tools, average execution for contract processing takes time of 1 minute 24 seconds per contract [18]. For this research, during the initial experiment scope, Mythril was considered as a tool of choice for comparison, to be evaluated against the multi-agent model, but due to high noise and inability to process older smart contracts' compiler versions the choice fell on other tools.

Slither performs static analysis by extracting the contract's structure and CFG (Control Flow Graph), a representation of all possible execution paths, from the Solidity AST (Abstract Syntax Tree), which is a structured tree representation of the source code that captures its syntactic elements in a hierarchical form without formatting detail. This is then translated into SlithIR, an internal representation of SSA (Single Static Assignment), where each variable is tracked and assigned only once. The refined data is then processed through a series of pre-defined analyses to identify vulnerabilities and provide detailed contract insights [19].

As for limitations Slither, demonstrates low accuracy, inability to identify Bad Randomness and Short Addresses vulnerability classes by design [18]. However due to its ability to process smart contracts in a timely manner, in seconds, and the ability to adjust the amount of vulnerability checks in the report, which removes the noise and unnecessary for the experiment information. Such information, as an example, optimizational, informational and low criticality checks. Hence the choice fell on Slither for the experiment.

Manticore [15] is a tool that utilises a high-semantic-awareness analysis technique that identifies path predicates to generate precise inputs for exploring program states, ensuring that all discovered paths are reachable in concrete execution with no false positives [20].

However Manticore proved to be the least effective tool among the all evaluated above. It takes 24 minutes 28 seconds to process the smart contract, additionally facing timeouts during 30 minute executions and high resources consumption [18]. It showed lowest accuracy of 11% and inability to detect Bad Randomness or Short Addresses categories which overall positioned it as an impractical choice for large-scale smart contract security analysis [18]. In addition Manticore is no longer internally developed and maintained [15].

While these traditional static analysis tools provide an initial and foundational layer of security, they often struggle with high false-positive rates, information noise ratio, low accuracy, and an inability to process complex contracts logic. This gap and limitations allowed to extend the security by integrating Large Language Models into the smart contract vulnerability detection. With natural language processing power and ability to use datasets for training on vulnerabilities and patterns recognition, Large Language Models offer a change towards a more adaptable and secure blockchain ecosystem, where vulnerability detection could be met with higher accuracy rate against static analysis tools.

2.2 Dataset for smart contracts

During the initial research and experiment build for this work, the focus was also on which dataset to use. Different research papers in this field utilised varying datasets for benchmarking. For instance, SmartBugs Curated was used in [21], Code4rena in [22] and ScrawID in [23]. Other papers would use their own custom dataset, be it outsourced contracts from the blockchain directly, constructed by hand, sourced from biggest hacking cases or a combination of points above.

SmartBugs Wild [24] is a dataset containing 47518 smart contracts scraped from Etherscan [25]. These contracts do not have any labels or any ground truth. This dataset could prove useful for evaluating tool scalability and performance at scale, but it cannot be used to evaluate tools accuracy due to missing labels and inability to construct a source of truth for vulnerability detection.

ScrawID [26] consists of 6780 real world Ethereum smart contracts which are tagged with known security vulnerabilities. It was created to serve as a standard benchmark for evaluating smart contract vulnerability using any detection tools and methods.

The labeling was done using Slither, Mythril, Smartcheck, Oyente and Osiris, with majority voting to determine ground truth. It covers 8 vulnerability types including reentrancy, integer overflow, denial of service, and others [26].

SmartBugs Curated [10] is a dataset containing 143 annotated solidity smart contracts with manually tagged vulnerabilities across 10 vulnerability classes, designed to evaluate the precision of vulnerability detection tools [25]. Serving as base ground truth for comparison purposes. For this experiment SmartBugs Curated dataset was chosen, for the following reasons:

1) The dataset allowed to construct ground truth to be compared with.
2) The labeled data allowed to pinpoint exactly where the vulnerability was introduced in terms of row count and function name.
3) The size of the dataset is manageable for manual verification
4) The dataset is publicly available to use.

2.3 Large Language Model methods

Vulnerability detection utilizing the Large Language Model has evolved through different approaches with their own advantages and limitations. Such approaches are, in no particular order, but not limited to, **general knowledge** assessment [21], where the Large Language Model is tested on the knowledge that it has been trained on.

Prompting approach, that leverages instructions and constraints to guide the Large Language Model toward vulnerability identification [21].

Fine-tuning involves training the Large Language Model on a datasets revolving around specific tasks, where the datasets are labeled with vulnerable code samples. This is done to improve parameters of a Large Language Model and specializing in that particular task [7] [22] [27].

Multi-agent systems suggest role-specialized Large Language Models agents that in collaboration analyze, cross validate and assess the code through structured reasoning. In this case an agent is given tools for their underlying task, but the decision to use the tool for assessment falls under agents' reasoning, meaning the agent may use its own general knowledge to fulfill the task rather than relying on the tool [22], [27], [28].

In addition, there is a possibility to construct hybrid approaches, combining prompting, fine-tuning, and multi-agent coordination to leverage each strengths, aiming to improve the performance [7] [27].

However, as approaches evolve, they are met with limitations. Such limitations are attributed to dataset constraints, where the dataset could be too little for a Large Language model to be trained for a specific task. Or training the model on data with a specific time period, which prevents the model from operating on more recent information.

Computational resource requirements, where the experiment could require processing power or memory that exceed the experiment's budget or hardware capabilities, as an example, when evaluating large codebases or deploying multiple model instances [21].

Model token limitations, where amount of source code or contextual information that cannot be processed in a single prompt, potentially leading to incomplete analysis, process freeze or an error as an output. [21]

Hallucinations where a Large Language Model may generate or fabricate incorrect vulnerability reports or security issues that are not present in the analyzed code [7].

Training data contamination, where the dataset that is required to be used in the experiment for the training purpose may already be present in Large Language models' general knowledge. These limitations and more, continue to affect the reliability of vulnerability detection in smart contracts across multiple research papers.

2.4 State of the ART

In the existing research it is worth pointing out current "state-of-the-ART" and their approaches that show improvements in vulnerability detection and demonstrating performance above 90%. These studies rely on enhanced symbolic execution, graph based pre-training, or single-model fine-tuning pipelines. As contract complexity increases vulnerability patterns become more intricate to notice. Examples of it could be zero day vulnerabilities that would require tools to operate outside the constraints, hence there is growing interest in distributed architectures that can perform analysis in specialized subtasks. Such a shift in perspective has motivated the current research to explore the multi-agent, tool-augmented framework for smart contract security.

SMARTEST [5] found 93.0% of known Common Vulnerabilities and Exposures (CVE) vulnerabilities in sampled contracts versus Mythril's 37.2% The model was trained on two main datasets, CVE dataset with 443 smart contracts containing arithmetic vulnerabilities and leaking-suicidal dataset with 104 contracts.

The approach was a language model guided symbolic execution combined with enumeration of transaction sequences with statistical n-gram language models (3-gram) trained on vulnerable transaction sequences. Using this approach the model learns probability distributions over vulnerable sequences from training data, and uses these probabilities as a cost function to prioritize symbolic execution paths most likely to reveal vulnerabilities. This approach allowed to discover seven zero-day vulnerabilities.

Peculiar [6] achieved Precision 91.80%, Recall 92.40%, F1 92.10% used SmartBugs Wild Dataset: 47,398 sol files with 203,716 contracts. The dataset was split as 20% for training and 80% for testing. The model was pre-trained utilising Crucial Data Flow Graph (CDFG) for Vulnerability Detection.

And fine-tuned LLM-Agents, iAudit [7] F1 91.21%, Accuracy: 91.11%. The model was a two-stage fine-tuning (Detector and Reasoner) with LLM-based agents (Ranker and Critic) using LoRA on CodeLlama-13b model, trained on 1734 positive samples and 1810 negative samples, benign code samples. The dataset was split as 2268 contracts for training, 567 contracts for validation and 709 contracts for the testing phase.

2.5 Methods of multi-agent Frameworks

To get an overview of current developments in the field of multi-agent systems for vulnerability detection in smart contracts, a search and review of relevant literature papers related to this topic was conducted. This review enabled to identify contributions and the approaches proposed in this field. These approaches, proposed by the papers, differ significantly from each other, providing distinct methods, ranging from classical multi-agent role based coordination models to

conversational LLM-based collaboration and fine-tuned hybrid frameworks, while addressing the same problem.

One research, LLM-BSCVM [27], has explored multi-agent systems as a mechanism to enhance Smart contract auditing through a three stage process, “Task Decomposition” for vulnerability detection, cause analysis, repair suggestion generation, risk assessment, vulnerability repair, and patch evaluation. “Knowledge Retrieval”, where the vulnerability knowledge base and external data sources are dynamically retrieved to enhance the model’s understanding of vulnerability causes and repair strategies. And “Result Generation”, where the agents, in conjunction with the retrieved relevant knowledge, generate vulnerability detection reports, cause analyses, repair suggestions, and final audit reports. The framework employs multiple specialized agents supported by retrieval-augmented generation (RAG) and fine-tuned CodeLlama models. “The RAG enables the model to dynamically retrieve documents from an external knowledge base while processing the input, merging the retrieved information with the input, thereby enhancing the accuracy and reliability of the generated results” [27].

While it demonstrates strong detection performance and reduced false positive rates, the approach requires supervised training and external knowledge base maintenance, increasing architectural complexity and computational cost.

The framework presented in SPEAR [28] constructs a distributed auditing system consisting of five agents responsible for planning, execution, repair, command execution for safety monitoring, and coordination agent for resource allocation and conflict mediation. Each agent maintains local beliefs and communicates through structured message-passing protocols. Static analysis tools such as Slither and Mythril are executed within sandboxed environments, while Echidna is used for fuzz testing. The primary contribution of this work demonstrates how distributed coordination improves resilience under failing scenarios. The evaluation is focused on system robustness and quantitative vulnerability detection accuracy on standardized benchmarks. However, it is not explicitly stated if the paper is focusing on vulnerability detection. In addition, the paper relies on a relatively small dataset, consisting of 16 scenario cases and furthermore, the tools seemed to be integrated into the workflow rather than being invoked dynamically.

Another framework SMARTIFY [29], is a multi-agent framework exploring vulnerability detection and repair in smart contracts written in Solidity and Move languages. By utilizing fine-tuning and Retrieval Augmented Generation (RAG) the system analyzes the code patterns and then generates code repairs. Multi-agent approach coordinates workflow that involve auditing, planning repairs, generating code, refining outputs, and validating the final result. The performance is evaluated using curated datasets from Solidity contracts, specifically from Trail of Bits [30] and Move based smart contracts. The effectiveness of such a system is based on the number of vulnerabilities fixed and the average inference time.

In contrast, LLM-SmartAudit [22] proposes a conversational multi-agent framework focusing on role specialization and action execution. Such architecture utilizes two analysis strategies Broad Analysis (BA), which aims to maximize vulnerability coverage, and Targeted Analysis (TA), which performs specific reasoning depending on vulnerability type. Agents, enhanced with a "buffer-of-thought" mechanism, assume specialized roles, such as code comprehension and vulnerability reporting, and collaborate through iterative conversational feedback loops. The system is evaluated on labeled benchmark datasets and real-world Code4rena [31] contracts, reporting high CVE detection rates and accuracy compared to traditional tools. Nevertheless, the architecture remains fully LLM-based, without integration of traditional static analysis tools, and computational cost due to iterative communication is acknowledged as a practical limitation.

Another direction is represented by IAudit [7], which combines supervised model fine-tuning with multi-agent integration. The proposed two-stage framework first fine-tunes a "Detector" model to identify vulnerable contracts and a "Reasoner" model to generate explanations. These explanations are then evaluated through a structured debate between two specialized agents such as "Ranker" and "Critic", that discuss and select the most plausible vulnerability causes. The paper utilizes CodeLlama-13b model and achieves F1 and accuracy score above 90%.
This paper's methodology follows a perception first and then analysis pipeline. For model training, the 1734 vulnerable and 1810 benign contracts were curated from auditing reports. The research evaluation was conducted on a separate set of 263 real world vulnerabilities.

Although the reviewed papers provide insights into the specifics of the design methods that were utilized, several common patterns become apparent. As an example papers with high-performing systems, when addressing the problem, rely on supervised fine-tuning of foundational Large Language Models.

Other multi-agent conversational system architectures would often operate without the integration of standardized static analysis tools. And comparisons against static analysis tools are not explored. Additionally, the direct comparison of performance evaluation between multi-agent and a single agent or a Baseline, under identical conditions are not addressed.

The papers that utilize multi-agent structure and follow tool-augmented approach do not explicitly specify if the tools are dynamically invoked or used as a mandatory component of the decision-making process.

Few studies would evaluate multi-agent within experimental setup, however the dataset, that is outsourced or curated for the research, is not publicly available, created internally or archived. This limits the reproducibility of the experiment, making independent validation challenging.

As a result, these recurring limitations highlight several gaps in the existing literature that require further exploration. Hence, this work investigates whether role specialized agents augmented with tools can improve vulnerability detection under controlled conditions without modifying model parameters through fine-tuning.

2.6 Novelty

Research papers in smart contract vulnerability detection field that utilize multi-agent systems explore various methodologies, from enhanced symbolic execution, graph-based pre-training, or fine-tuned Large Language Model trained on curated vulnerability datasets. This research focuses on a role specialized, tool-augmented multi-agent architecture operating under controlled experimental environment, rather than developing new model training technique or approach.

The primary novelty of this research lies in the orchestration of the multi-agent Large Language Model framework augmented by the static analysis tools and evaluated under a controlled, reproducible benchmark dataset. The vulnerability detection task is evaluated by three perspectives: A Smart Contract Developer agent performing knowledge driven reasoning without tool assistance. A Smart Contract Security Specialist agent augmented with Slither static analysis reports. And a Smart Contract Auditor agent leveraging SolidityScan reports to support decision making.

Another contribution of this research is the direct comparison of three configurations under identical conditions using the SmartBugs Curated dataset: A baseline in form of a standalone Claude Large Language Model, a role-specialized multi-agent architecture, and the Slither static analysis tool. Such comparative evaluation provides insight into whether multi-agent reasoning and selective tool augmentation can produce competitive results in realistic deployment scenarios. In addition, this research analyzes whether agents utilize external tools during the decision-making process, evaluating the success rate of such interactions, and comparing enforced tool usage with voluntary tool invocation.

To the best of author's knowledge, existing literature does not address direct, controlled comparison between single Agent Large Language Model reasoning, tool usage in tool-augmented multi-agent Large Language Model orchestration, and classical static analysis within the same benchmark framework without fine-tuning. Therefore, the novelty of this research lies in its systematic evaluation methodology, analysis of tool usage, and architectural comparison. The key distinctions of the current research and reviewed papers are summarized in Table 1.

Table 1. Comparison of Slither, Baseline and multi-agent performance.

Feature	LLM-BSCVM	SPEAR	SMARTIFY	LLM-SmartAudit	IAudit	Current Research
Architecture type	MAS RAG fine-tuned	Distributed MAS coordination	MAS RAG fine-tuned	Conversational MAS	Debate-based MAS reasoning	Role-based tool augmented MAS
LLM fine-tuning required	Yes	**No**	Yes	No	Yes	**No**
Static analysis tools integrated	No	Slither, Mythril, Echidna	No	No	No	Slither, SolidityScan
Tool augmented reasoning	Partial (RAG)	**Yes**	Partial (RAG)	No	No	**Yes**
Dataset used for evaluation	Dappscan reports	Not standardized benchmark	Curated Solidity datasets	Code4rena and labeled datasets	Custom curated dataset	**SmartBugs Curated**
Dataset publicly available	Partially	Not specified	Partially	Yes	No	**Yes**
Dataset Size	263 smart contract 1199 reports	16 scenario cases	60 solidity 92 Move contracts	6454 smart contracts	709 smart contract	142 smart contracts
Comparison with Baseline	Yes	No	No	No	No	Yes
Comparison with tools	No	Yes (Slither, Mythril)	No	No	No	Yes (Slither)

3 Methodology

With a formulated hypothesis, which states that a role-based multi-agent Claude system version 4.5 Sonnet, will achieve higher detection accuracy and lower false positive rates compared to a static analysis tool, this work has implemented a quantitative research method.

As quantitative research requires to discover facts by measurement and data analysis, this method was best suited for the experiment where the performance of two independent studied variables were investigated. The study has implemented an experimental research approach and the experiment was evaluated with controlled variables using traditional scientific methods.
For scientific methods, data analysis and performance evaluation were calculated, where performance was based on standardized True Positive, False Positive, True Negative and False Negative attributes.

3.1 Performance attributes

Each of the attribute is distinct and represents the following:
True Positive is an attribute that is counted in cases where the vulnerability was present in the provided smart contracts and that vulnerability was successfully detected according to the label in the ground-truth file. This attribute is not represented in the benign data, because non-vulnerable smart contracts do not have vulnerabilities by default.

True Negative is an attribute that is counted in cases where no vulnerability existed and was not detected according to the label of the ground-truth file. For the benign data this attribute is inverse of True Positive for vulnerable data. Because benign data has no vulnerability, True Negative shows correct identification of a clean contract that has no issues.

False Positive is an attribute that is counted in cases where the vulnerability was incorrectly detected as one or more vulnerabilities that have no corresponding label to ground-truth file.
For both vulnerable and benign data this attribute represents that the components found something that did not exist in the smart contract.

False Negative is an attribute that is counted in cases where the vulnerability was present in the processed smart contract but was not detected by the components according to the ground-truth vulnerability class. This attribute is not represented in the benign dataset as the vulnerabilities do not exist in the clean smart contract.

It is worth pointing out that in this experiment, True Positive was recorded on two occasions. First, vulnerability should be found in the smart contract, correctly identified with the class according to the ground-truth and correct vulnerable function should be reported. Another case where True Positive was recorded is in addition to correct identification of a vulnerability class, the correct vulnerable row was stated. If the component would identify one of these two cases this would suffice to flag contracts as True Positive.

In addition, if the final output would not match the vulnerability class of a ground truth, this was considered as False Positive and False Negative. The reason behind is that the mismatching vulnerability class means that the correct vulnerability was not detected which is False Negative and an incorrect vulnerability was reported which is False Positive.

Each reported result was manually inspected against the ground truth file. With the attribute in place it was possible to calculate the performance metrics of each component of the experiment.

3.2 Research Design

The research was aimed to evaluate the performance of Slither, a traditional static analysis tool for smart contract vulnerability detection and a multi-agent LLM framework, including forced and selective tool usage, and baseline. In the remainder of this research, the multi-agent framework, including selective and forced approach, the baseline, and Slither, the static analysis tool, are collectively referred to as “components.” The performance of each component was calculated based on True Positive, False Positive, True Negative and False Negative attributes and compared against each other. The results of the experiment are then presented. Additionally the limitations that were encountered during this scientific experiment were noted and pointed

out. The research design is based on the data, Slither tool and multi-agent Framework, that is discussed further on.

3.2.1 Data format and filtration

For the experiment to run, it should be based on data. To evaluate the performance of the experiment the SmartBugs dataset [10] was chosen. This is a standardized dataset which is publicly available, making it easily available for results replications. The Smartbugs dataset consists of ten smart contract vulnerability classes with different amounts of smart contracts in the dataset which are also annotated with the vulnerability class and exact row where vulnerability was present. The following vulnerability classes are defined based on the SmartBugs dataset [10]:

Reentrancy with 31 examples - occurs when external contract calls are allowed to make new calls to the calling contract before the initial execution is complete.

Unchecked Low Level Calls with 52 examples - occurs when the return value of low-level calls such as call() or send() is not checked, allowing silent failures to go unhandled.

Access Control with 18 examples - occurs when one can access a contract's functionality through its public or external functions.

Arithmetic with 15 examples - occurs when integer overflow or underflow is not handled, leading to unexpected computation results.

Bad Randomness with 8 examples - occurs when predictable variables are used for random number generation, allowing attackers to influence outcomes.

Short Address with 1 example - occurs when insufficient input validation allows malformed addresses to be passed, causing the EVM to pad the input and misinterpret token amounts.

Denial Of Service with 6 examples - occurs when an attacker can permanently block contract execution, preventing legitimate users from interacting with it.

Front Running with 4 examples - occurs during a pending transaction where the attacker submits their own with a higher gas fee to execute first and gain an advantage.

Time Manipulation with 5 examples - occurs when a contract relies on block.timestamp for critical logic, which miners can manipulate.

Other with 3 examples - vulnerabilities that do not fit the above categories.

Each contract was inspected manually and picked for the experiment. Smart contracts that did not have any code or were interface extensions, which also do not have any code lines, were not present in the dataset for this experiment, as such contracts do not bring any value in performance evaluation.

In addition to the vulnerable dataset, the **Benign** dataset with 30 examples was also introduced to the experiment. For benign data OpenZeppelin, which is an industry standard for smart contracts written in Solidity and other languages [11] was used in this research. The data filtration for the benign data was the same as for the vulnerable - only the smart contracts with a meaningful amount of code were used in the research. This allowed to source 30 benign smart contracts that represent a good portion of the overall dataset.

Based on this dataset the ground truth file was created, where each smart contract was annotated and was ready to be compared against results of each component of the research. Overall the dataset consists of 30 contracts of benign data and 143 of vulnerable data, making the total dataset of 174 smart contracts.

3.2.2 Slither workflow

Slither tool was executed locally by installing it via the GitHub repository [14]. The process was such that each contract had to be processed one by one, manually, using the command line inside PyCharm [32], an Integrated Development Environment for software development.
Running contracts one by one was done due to the fact that batch processing failed for smart contracts that were written in older versions of Solidity. Slither could not process newer and older versions at the same batch, hence for each failed smart contract, Slither compiler version had to be installed and reprocessed accordingly. Additionally, a Slither output report was constructed in such a way that optimizational, informational and low criticality checks were dropped. This allowed saving of resourcing power in terms of API input tokens and affected smart contract processing time when reports were used by the multi-agent framework.

Depending on the size of the smart contract the time for the process also varied, making the longest process to be around 10 seconds and the shortest around 2 seconds. As a report by Slither was created, the vulnerability checks were extracted for each report and noted.

The vulnerability check is a confirmation by the tool that a particular vulnerability exists in this smart contract. Slither vulnerability check and SmartBugs dataset vulnerability classes were mapped together. It was done by cross referencing the Vulnerability Mapping project [33] and cross checked with Detector documentation project [34]. The exact mapping can be viewed under the Appendix 1.

3.2.3 Multi-agent workflow

The multi-agent framework was orchestrated by the n8n service that allowed the creation of multiple agents in a coherence environment to automate workflow and decision-making output. Each agent underneath was utilizing Antropic's Claude Sonnet 4.5 model, which was the latest model at the time of writing. To communicate with the models an Antropic's API service was used, which was then connected to n8n service via API token key.

To evaluate the performance of the multi-agent framework, the same SmartBugs curated dataset was used, that we also used to create a ground truth file. However, in the case with LLM the test

dataset had been formatted in a way that the code pointing to the exact row where the vulnerability exists, was removed. Meaning that the smart contract only consisted of functional code, without any comments or hints. This was done to remove any bias for the multi-agent framework to determine if vulnerability exists in the proposed smart contract.

The multi-agent framework consisted of three agents, each with its own distinctive role. In the scope of this research three particular roles were chosen that are closely related to coding, vulnerability detection and blockchain technologies. The roles were appointed as Smart Contract Developer, Smart Contract Security Specialist and Smart Contract Auditor.

Smart Contract Developer was given a scope where it was an expert in Ethereum Smart Contracts, specializing in Solidity and blockchain development. The characteristics for this agent were set as abilities to write secure, efficient, and gas-optimized smart contracts, and knowledge of static analysis tools, the knowledge of implementing common patterns for smart contracts such as ERC-20, ERC-721, ERC-1155 standards that define how tokens behave and interact across the ecosystem [35], and understanding blockchain architecture and consensus mechanisms. Additionally the knowledge of best practices for contract deployment and upgrades were set. However no tools were provided to this agent as it had to rely on the general knowledge to determine if vulnerability exists when processing smart contracts.

Smart Contract Security Specialist scope was deep expertise in blockchain security and included following characteristics: the ability to identify common vulnerabilities, understanding attack vectors and exploit patterns in smart contracts and the knowledge of smart contract best security practices. For the tool, the agent was provided with the Slither static analysis tool, more precisely the report that Slither produced during evaluation. To stay consistent with Slither, the agent was provided the same vulnerability mapping between Slither and SmartBugs. This allowed to remove any hallucinations related to non-existent vulnerability in the dataset.

Smart Contract Auditor scope was to review the smart contracts in the audit perspective, with general knowledge and best practices. For the tool the agent utilized a SolidityScan [36] service report to determine if vulnerability exists. Each contract was inspected by the SolidityScan

service and generated a unique Uniform Resource Locator (URL) that contains the report of the service with its own vulnerability checks. The agent was provided with the report and the smart contract corresponding to the report to make the decision. The vulnerability checks of SolidityScan are self explanatory and correspond to the SmartBugs dataset vulnerability class, meaning no cross mapping was needed in this case.

Before the experiment could run, the input and output were also configured on the whole framework level. The input consisted of instructions to analyze the provided smart contract and reports provided by Slither and SolidityScan, if applicable to the agent. Meaning that each agent would process and analyze the provided smart contract, but only the Smart Contract Auditor would additionally analyze the SolidityScan report and make the decision. And only Smart Contract Security Specialist would utilize Slither report for decision making. From then the expert judgment should have been used by the agents. To stay in scope and remove potential hallucinations these instructions were marked as critical to follow.

For the output the results were expected to be only in json format, the description of where vulnerability exists should have been ‘Yes’ or ‘No’ with no more than 2 sentences or 50 words of explanation. This was done due to earliest iterations of experiments where by allowing to create output reports without word limitations the multi-agent framework would generate a comprehensive report and spend an excessive amount of API tokens.

In addition, a second round of experiments was conducted. The only difference in the architecture was the introduction of selective tool usage by the agents. In this scenario, all agents were provided with access to the same set of tools, while having the ability to decide whether to invoke them during the decision-making process or not. This approach allowed measuring the tool invocation frequency for each agent by denoting each decision with binary indicator TRUE/FALSE. Furthermore, it allowed for the collection of additional statistics related to the success and failure rates of vulnerability detection with and without tool assistance for each agent. Finally, the experiment evaluated the performance differences between enforced and selective tool usage strategies.

For the optimization of Slither output, the report was constructed such that optimizational, informational and low criticality checks were dropped, this reduced the amount of information in the report. SolidityScan reports did not require optimization as they were provided in such a way where only the vulnerability checks were present. This is done due to smaller reports allowed to save API tokens during the smart contract processing stage by the multi-agent framework, as each agent would process the same contract. As the Large Language Model token consists of approximately 0.75 length of the word, the bigger the input in terms of words, the more tokens needed to be spent on the processing power.

The list of vulnerability according to Smarbugs Dataset was provided to the agents, this was done to remove any hallucinations and stay consistent with Slither and SolidityScan vulnerability mapping. In case if multiple vulnerabilities were identified only the most critical should have been reported by each of the agents.

The entire process was divided into multiple steps. First, the code of the smart contract that required processing, was provided as an input, the smart contract at this point is functional code without any comments, the reports from Slither and the SolidityScan corresponding to the processed smart contract were also provided. As soon as processing started the Smart Contract Developer would check the code and give his decision with reasoning, then the Smart Contract Security Specialist would scan the smart contract and utilize Slither's report for decision making. Finally, the Smart Contract Auditor would scan the smart contract and utilize the SolidityScan report to decide on the result. The workflow for the forced tool is represented in Figure 1 and configured as follows: the input with reports are provided, next node assigns input into different variables. Then these variables are distributed to the agents with appropriate reports. On the next step each agent processes the input independently and in parallel. Then all results are gathered and formatted. On the output if the vulnerability is chosen by consensus, the final node will give the result. In case if there is no consensus, all information is transferred into a gathering node which then will start a new round of discussion by providing all information back into input. Such an approach allows one to see the reasoning behind each agent and its role where deterministic findings of static analysis tool are integrated into agentic decision-making processes.

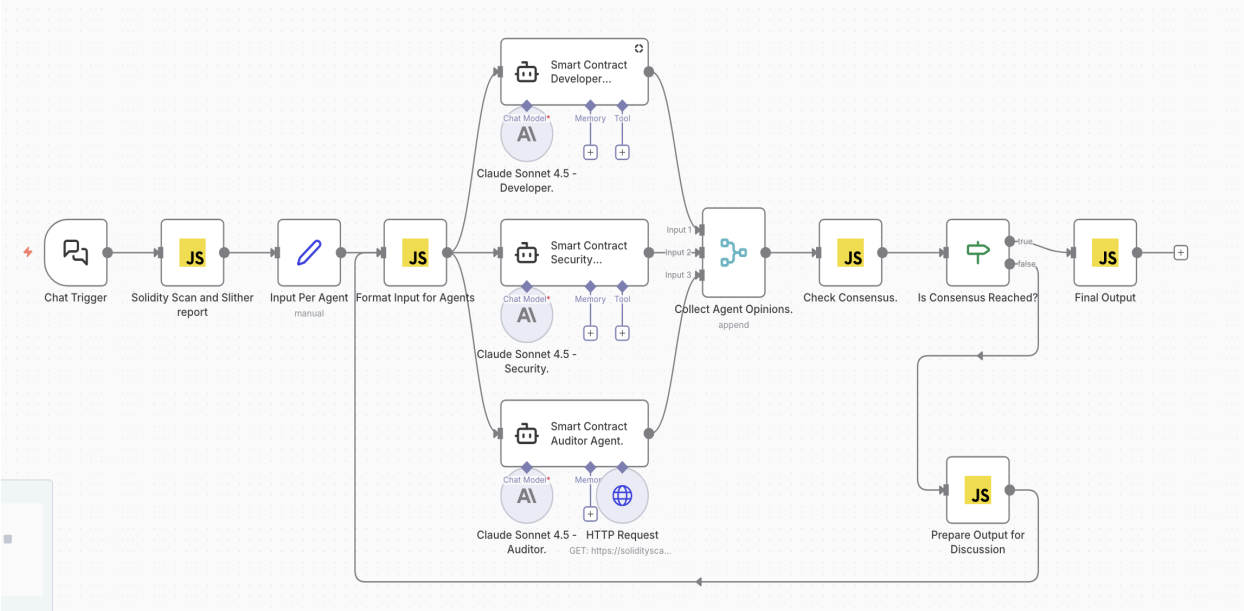

Figure 1: Multi-agent forced tool usage workflow

In the case where all agents concluded the same vulnerability class, the results were documented against the ground truth file. The majority voting approach was used in the cases where two out of three agents would decide on a particular class, then the majority dictated the vulnerability class decision. However, in cases where all the agents did report different vulnerability classes, another iteration of discussion took place. In this step, all previous reasoning in the form of 2 sentences or 50 words from each of the agents were provided as additional input with the smart contract's code. This allowed the agents to utilize previous information for better decision making. It is worth making a note that out of 174 smart contract scans, only 12 instances required discussion reiteration. This experiment never went above 2 iterations, which could hint on high consensus among the agents.

For the selective tool usage workflow, depicted in Figure 2, the workflow has the same steps as before. The only difference is the introduction of code blocks after each agent. Each such code block contains a function to invoke the usage of reports that are appropriate to each agent. If a function was invoked then final output would mark that the tool was used with a TRUE/FALSE flag.

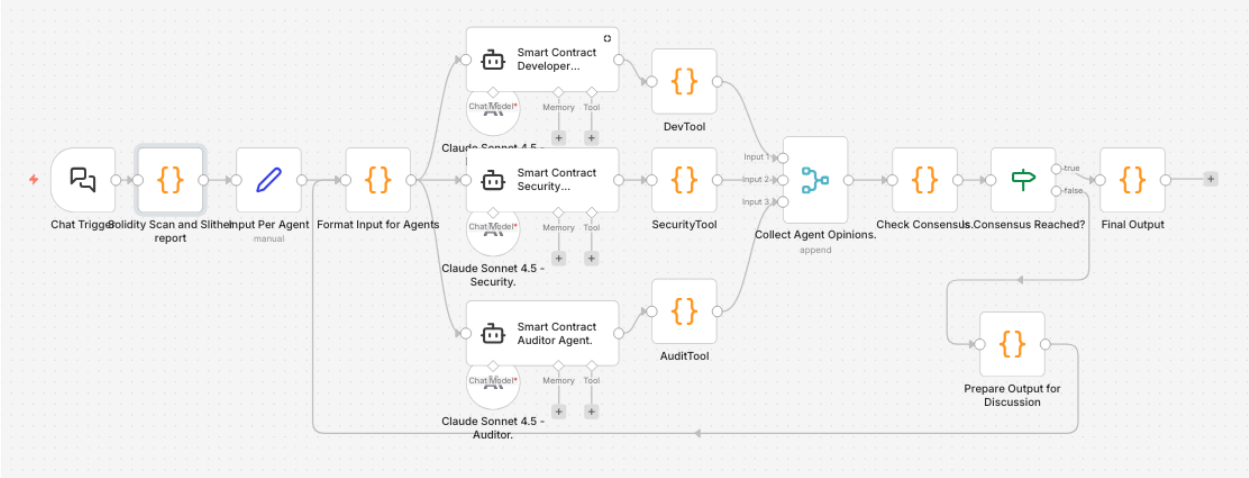

Figure 2: Multi-agent selective tool usage workflow

It is worth making a note around API token cost requirements. The computational cost of operating a multi-agent vulnerability identification and analysis framework is dictated by token consumption through the Claude API service. Processing a single smart contract on average takes approximately 1200 tokens, the average is based on the smart contract that are introduced in the SmartBugs dataset, meaning a real life contract with extensive amount of code would cost more to be processed. Alongside the smart contracts, the provided Slither and SolidityScan reports would be major contributors to token consumption.

To balance it out, the API token consumption would also depend on the amount of information chosen to be omitted. In this experiment the information related to the optimizational, informational and low criticality checks were removed from the reports. This allowed to cut the operation cost, that would require at least double the amount of tokens per Slither report on average.

SolidityScan reports, however, had no information filtering and thus were provided as is. This would result in 1350 tokens on average per SolidityScan report and agents processing power combined. Overall the whole input would require the SolidityScan report, Slither report, processing resource per agent, distributed across 3 agents with additional 65 tokens for output per agent, as the results are constrained with no more than 2 sentences or 50 words of

explanation for the final output. Overall, in a multi-agent architecture with three agents, on average consume 1700 tokens for Security agent with Slither report, 1350 tokens for Auditor agent and SolidityScan report and 1230 tokens for Developer agent. This sums up to a total of 4280 tokens in total for one processing cycle with additional input tokens and 195 (65 per each agent) output tokens per round of discussions.

However, when agents engage in the iterative rounds of discussion to re-evaluate vulnerability class, the token cost multiplies, as each discussion round requires re-submitting the same vulnerability reports, smart contract code to all participating agents and the reasoning behind each agent, which increases the input token consumption.

For instance, a two round discussion process would consume approximately $8755\ (4280\ *\ 2\ +\ 195)$ input tokens and 195 tokens as re-evaluated output results from each agent, this effectively doubles the computational cost. Using Claude Sonnet 4.5 pricing of 3 USD per million input tokens and 15 USD per million output tokens, a single smart contract processing cycle would cost approximately ~0.013USD $(4280\ *\ 3/100000)$ for input, while multiple discussion rounds require twice more per contract. The API token consumption becomes relevant when scaling, to analyze larger smart contracts code-wise or optimizing the reports provided by the tools to the agents.

3.2.4 Baseline

For the baseline the same Claude model Sonnet 4.5 was chosen. The model was processing smart contracts one by one using Anthropic's User Interface (UI). The model was given similar instruction to the agents, apart from any specific abilities and tool usage. Meaning that the model had to decide if the smart contract is vulnerable “Yes” or “No”, what class the vulnerability is related to, if it exists. And in which row or function name did the vulnerability occur. The list of vulnerabilities according to the SmartBugs dataset was also provided. The idea was to have a model equipped with general knowledge, whose performance could be compared against the multi-agent framework and Slither.

The input for the baseline followed the same principle as for the multi-agent framework. All of the smart contracts were preprocessed to remove any hints or comments. This ensures that the experiment would minimize, if not remove entirely, any bias for the model. The results were documented and compared against ground-truth file.

It is worth noting that standalone baseline lacks architectural rigor that multi-agent have. While baseline may process the smart contract in a single path reasoning, multi-agent framework separate the detection into different concerns such as code structure and security patterns. In addition, by distributing the analysis and implementing a majority consensus, it ensures vulnerability cross-validation, which increases reliability of the final output.

3.3 Ethical Considerations

This research evaluated and compared the performance of static analysis tool Slither, a multi-agent framework and a baseline model for smart contract vulnerability detection. Both the multi-agent and baseline utilized Claude model instances. The experiment relied on a curated SmartBugs dataset taken from Ethereum blockchain. The dataset itself is publicly available and accessible. During the experiment, there was no deployment of vulnerable code to the public, nor were any new vulnerabilities, including zero day vulnerabilities.
Additionally, the work did not possess, encounter or share any sensitive information. And no new tools that would harm the public were invented, hence no specific ethical concerns arose from this research.

4 Results and analysis

This chapter presents the evaluation results for the multi-agent framework, static analysis tool Slither and a baseline Claude model across the SmartBugs [10] dataset. Each smart contract was assigned a predefined vulnerability class, with an additional benign class included to reflect a realistic detection scenario. Performance of each component was measured using Accuracy, Precision, Recall, F1 score, which is the standard metrics for classification tasks in the vulnerability detection literature.

4.1 Performance metrics

The performance evaluations were calculated based on the gathered attributes after processing the dataset by each component. This allowed us to measure the quantitative metrics such as Accuracy, Precision, Recall and F1.

The recall metric measures how many of the vulnerabilities present in the dataset were correctly identified by the components. This metric shows the ability to detect relevant instances and is calculated as following:

$$Recall = \frac{TP}{TP + FN}$$

The accuracy metric shows how many total predictions were correct. Note that for vulnerable dataset True Positive is used for correctly identified cases and True Negative is used for correctly identified benign data cases. Accuracy is calculated as following:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

The precision metric indicates how many predicted positives were actually correct and is calculated by the following equation:

$$Precision = \frac{TP}{TP + FP}$$

F1 score shows the balance or a harmonic mean between precision and recall. It is based on recall and precision metrics and is calculated by the following equation:

$$F1 = \frac{2 * (Precision * Recall)}{Precision + Recall}$$

4.2 Performance Metrics

Micro-average aggregates True Positive, False Positive and False Negative counts across all classes before calculating metrics, giving equal weight to each instance. Such a metric is less suitable in an imbalance dataset as some vulnerability classes have more smart contract instances than others which may lead to incorrect results [37].

Macro-average calculates metrics independently for each class and then averages them, giving equal weight to each class regardless of its size [37].

For this reason current research utilizes the macro-average metric as the primary metric and all performance metric results are represented as percentages.

4.2 Performance Analysis

The performance for Slither, Baseline and multi-agent, including the selective and forced tool usage was measured across 10 vulnerability classes stated in the SmartBugs Dataset mixed with benign data. Due to the difference on how calculation is done on vulnerable and benign data, the performance of the vulnerable dataset is presented in the tables below, however the benign data is measured with the overall accuracy metric as the part of the whole dataset.

It is important to note that, although the dataset would introduce ten vulnerability classes for this experiment, the reported measurement results include only nine classes. Such a decision to omit, was made from the inability of all components to identify the Short Address vulnerability class. Due to its representation in the SmartBugs dataset as a single smart contract instance, this class was excluded from the performance evaluation tables. Nevertheless, it is still included within the macro metrics level to ensure overall performance evaluation.

4.2.1 Report data filtration

The reports provided for each of the agents originated from Slither and SolidityScan tools. The SolidityScan report had only the vulnerability checks in them. The process was such that a generated report would have an URL generated per each report, security checks were extracted from them and used further on in the experiment. However upon inspection of Slither reports, it was noticed that the reports contained information that would hint to the agent what contract name was processed. After all reports were generated by Slither, it was decided to minimize potential information leakage and to remove the bias from the agent. Each report was processed by a python script that would locate the metadata inside of the contract and remove tags that would be associated with contract original names. The names of the contracts were not removed from the report but were renamed into something unambiguous. With the same logic using the same python script, the second part of data filtration was to locate according to the report smart contract and rename the smart contract itself. This allowed to have a clear mapping of what smart contract is related to what report, without explicitly hinting what was processed. The vulnerable function name however stayed, otherwise the reports would not be beneficial to the agent's decision-making process. This process of report filtration was conducted due to the fact that Large Language Models are trained on the existing publicly available data. SmartBugs dataset that consists of annotated vulnerable smart contracts is free and publicly available. The reports related to benign data were processed exactly the same. OpenZeppelin is a project where clean smart contracts were sourced from. It is also publicly available and could be potentially known by the models just like SmartBugs dataset.

This approach was applicable to multi-agent frameworks, for forced and selective tool usage, and a baseline represented by a single Claude model. The main goal for such an approach was to minimize cases where input report data could already exist in the model knowledge base, however this approach does not guarantee a complete successful rate as the smart contracts code could also, possibly be in the knowledge base of a model. This has also been noted in the limitation section of current work.

4.3 Multi-agent frameworks performance

The multi-agent framework was divided into two setups: selective tool and forced tool usage, both setups consisted of three agents that are based on Claude model. If the result would end with a dispute between the agents, a new round of discussions would start. In this new round each agent would get the initial smart contract and output result of each agent. The whole experiment was constrained to up to three iterations of discussions for consensus. The difference between forced and selective tool usage approaches lies in how the tools were integrated.

In the forced setup agents were provided with a static analysis tool's reports as part of the input in the workflow, to decide on the vulnerability class. This means that the agents use reports as a mandatory component of the process for their decision-making output. Each report was assigned into a variable and that variable would then be assigned per each agent. Such an approach would restrict the visibility of each report to only a particular agent that would be assigned to that variable.

For selective tool usage setup, the reports were provided to the agents as the JSON (JavaScript Object Notation) format. JSON is a format with data interchange format that consists of name–value pairs in a human-readable format. The report in JSON format could be then invoked by the function. Each such function with JSON formatted report assigned to each agent. The process is similar to forced tool usage, where each report is available to a particular agent, according to his role. By choosing to invoke the function each agent could decide whether to use the report during the decision-making process. This allowed to calculate failure and success rate in accordance with the tool usage.

In such cases where the report would not consist of vulnerability checks, meaning no issue was found by the tool or being purely empty, the report was considered as clean and still provided to the agents. These cases were especially relevant when processing benign smart contracts.

4.3.1 Multi-agent with forced tool usage performance

As soon as workflow was constructed using n8n service and API token key was successfully connected to it, agent became available for the performance evaluation. The input included three parts. The first part was the Slither report, which was filtered out as mentioned previously. The second part was the SolidityScan report and the last part was the smart contract itself.

The processing started with assigning each part of the input into a separate variable that was applicable to each of the agents, according to its role. Then the processing took place. Each variable would be accessed as part of the workflow and decision making. That would ensure that the variables were mandatory for the agents and it would enforce the usage of the reports. The first agent to receive it was the developer agent. As an input, the agent received smart contract only, as soon as the decision was made the process would be started for security specialist agent who would receive smart contract code and Slither report. As soon as the processing cycle finishes the new cycle would start for the Auditor agent. Such workflow where each agent processes the input one by one would remove the cases where the decision-making could be in parallel. This was done if due to potential incorrect coding or human error, the models could see each other's responses and become biased.

The responses were inspected manually and recorded alongside the ground truth file. The correctly identified vulnerability class would be if the correct vulnerable name or the row was provided. The results were calculated based on the attributes and metrics explained above.

During the forced tool usage experiment, there were cases of agentic dispute recorded. Out of the dataset that consisted of 173 smart contracts, 12 times the agent did not come to consensus. This represents 6.9% of disagreement, the rest 93.1% include cases where all three or two out of three agents agreed on a common vulnerability class. Cases where consensus was not agreed upon

were reprocessed. This included the same input as previously, smart contract and variable with reports, but here a new input was introduced, a final decision of each agent. Each agent in this case would see what the other agent decided and it could affect the new decision. After 12 cases were reprocessed the output was recorded and it was final for performance evaluation. These 12 cases did not lead to additional disagreement, this leads to believe that such approach has a high agreeability between the agents.

With attributes in place the performance metrics were calculated. As seen in Table 2, the multi-agent framework was able to achieve a high chance of success in detecting Arithmetic and Reentrancy vulnerability classes.

Table 2. Multi-agent with forced tools macro average performance per classification.

Vulnerability class	**Accuracy**	**Precision**	**Recall**	**F1**
Access Control	0.89	0.94	0.94	0.94
Arithmetic	1	1	1	1
Bad Randomness	0.78	0.88	0.88	0.88
Denial Of Service	0.71	0.83	0.83	0.83
Front Running	0.33	0.5	0.5	0.5
Other	0.2	0.33	0.33	0.33
Reentrancy	1	1	1	1
Time Manipulation	0.25	0.4	0.4	0.4
Unchecked Low Level Calls	0.51	0.67	0.67	0.67
Benign	0.80	0	0	0

Based on the performance per classification, it was possible to calculate micro-average metrics related to Accuracy, Precision, Recall and F1. The results are shown in Table 3. However, for the micro-average the Accuracy metric is not applicable, as it is not assumed to be balanced. As the experiment is based on a fixed set of classes and gives equal importance to each class, micro accuracy is an unsuitable metric where vulnerability class distributions are represented by an

unequal amount of smart contracts in the dataset. Meaning that some vulnerability classes have more smart contract instances than others. This can lead to incorrect results due to rare classes, such as Short Address which is represented by only one smart contract.

Table 3: Micro-Average metrics

Micro-Average	
Accuracy	0.692
Precision	0.772
Recall	0.804
F1	0.788

Instead, macro accuracy is adopted. Based on TP, TN, FP, FN attributes, the calculation of macro average, including the accuracy, aggregates results over all instances and reflects the model's overall performance more reliably in this case where the dataset is imbalanced for each vulnerability class. This makes macro accuracy a more appropriate and representative metric for evaluating the multi-agentic framework in this context. The macro average performance metrics are presented in Table 4.

Table 4: Macro-Average metrics

Macro-Average	
Accuracy	0.568
Precision	0.656
Recall	0.656
F1	0.656

Although metrics show capability of the model in discovering vulnerabilities, they are still not near the state-of-the-art performance metrics. In this work no additional training or fine-tuning was part of the experiment. The agents were provided with clear instructions to assume their role and utilise the tools as part of the process. In the instances where deployability is a most important factor, this approach is applicable. Applicable because of ease of integration and being

cost effective in exchange for better recall metrics in comparison to static analysis tool presented further on. Recall metric measures how well the model is able to detect all relevant cases without missing them. It is an important metric because it reflects the model's ability to minimize missed relevant instances, ensuring that important cases are detected even if this may come out as increased false positives attributes.

4.3.2 Multi-agent with selective tool usage performance

The workflow for Multi-agent with selective tool usage was similar to Multi-agent with forced tools selection. As soon as the input with Smart contract and reports were provided, the agent would start the decision-making process one by one. As the function was available for Security specialist agent and Auditor agent the final output was configured in such a way that the decision, if the report was used or not, was flagged with TRUE or FALSE boolean. However, such boolean was not applicable to the developer agent. This was due to the scope of this research, as the developer agent was configured only to use general knowledge of the model and no tools with reports were assigned to him. Each output was inspected manually and was marked against the ground-truth file. This allowed to calculate performance of multi-agents represented in Table 5.

Table 5: Multi-agent with selective tool macro average performance per classification

Vulnerability class	Accuracy	Precision	Recall	F1
Access Control	0.89	1	1	1
Arithmetic	1	0.88	0.88	0.88
Bad Randomness	0.78	0.88	0.88	0.88
Denial Of Service	0.71	0.83	0.83	0.83
Front Running	0.14	0.25	0.25	0.25
Other	0.2	0.33	0.33	0.33
Reentrancy	1	1	1	1
Time Manipulation	0.25	0.4	0.4	0.4
Unchecked Low Level Calls	0.39	0.56	0.56	0.56
Benign	0.83	0	0	0

Same as in the previous experiment, there were instances of disagreements between the agents. But in this case there were only 2 of such instances, of which out of 173 smart contracts dataset is only ~1.16% of disagreement, the rest ~98.84% as also previously stated, included cases where two out of three or all three agents did come to a consensus.

After calculating the performance per classification, macro-average and micro-average metrics were calculated. In the micro-average Accuracy is not applicable to the same reasons stated previously, due to not being balanced. The results for micro-average are represented on Table 6 and macro-average on Table 7 accordingly.

Table 6: Micro average metrics

Micro-Average	
Accuracy	0.639
Precision	0.730
Recall	0.755
F1	0.742

Table 7: Macro average metrics

Macro-Average	
Accuracy	0.537
Precision	0.612
Recall	0.612
F1	0.612

For the current approach with selective tool usage method, new data was acquired in terms of TRUE/FALSE booleans. These booleans represent if the function was invoked hence making the report available to the agent. These metrics were gathered and marked per classification and represented in Appendix 4.

The “Tool Used (True)” in the table specifies how many times the agent used the reports for decision making. “Tool Not Used (False)” shows the amount of time the agent would trust their decision based on their internal processes. “Missed with tools (True)” represents the amount of vulnerabilities that were misclassified or not found when the report was used. And “Missed without tool (False)” is the amount of vulnerabilities that were misclassified or not found when the report was not used. After statistical aggregation the total amount of each instance is presented in Table 8.

Table 8: Instances of tool usage vs missed vulnerabilities

	Security agent	**Auditor agent**	**Developer agent**
Tool Used (True)	132	78	n\a
Tool Not Used (False)	41	95	173
Missed with Tools (True)	34	25	n\a
Missed without tool (False)	4	14	38

These metrics allowed further evaluation of impact of tool usage at the agent level. For the evaluation miss rating was introduced. Miss rate shows the proportion of instances where the agent failed to identify the correct outcome out of all instances, separately for “Tool used (True)” and ”Tools Not Used (False)” scenarios. Hence this metric is dividend for both scenarios as “Miss rate True” and “Miss rate False”. Such data aggregation provides a high-level view of how each agent performs under different tool usage conditions.

For the scenario, where tools were used the calculation would be the missed vulnerabilities under condition where tools were used divided by the total number of instances where the tools were used. In terms of equation it would look as following:

$$Miss\ Rate\ True\ = \frac{Missed\ with\ Tools\ (True)}{Tool\ used(\ True)}$$

A lower “Miss Rate True” condition indicates that tool usage improves agent performance, whereas a higher miss rate suggests that tool invocation is inefficient.

For the scenario, where tools were not used the calculation is the same, but using the number of missed vulnerabilities divided by the total number of instances where the tools were not used, in terms of the equation it looks as follows:

$$Miss\ Rate\ False\ =\ \frac{Missed\ without\ tool\ (False)}{Tool\ Not\ Used\ (\ False)}$$

A lower "Miss Rate False" condition indicates that an agent performs better without tools, whereas a higher miss rate suggests that decision-making process is inefficient.

As the Developer agent did not have access to tool usage, it could only contribute to the non-tool usage condition, hence only "Miss Rate False" under no tool usage is applicable for this case.

With aggregated data in place, the performance of how each agent under different tool usage conditions was calculated. The performance evaluation is shown in Table 9.

Table 9: Missed vulnerabilities with and without tools

	Security agent	**Auditor agent**	**Developer agent**
Miss Rate True	25.76%	32.05%	n\a
Miss Rate False	9.76%	14.74%	21.97%

The results show that a Security specialist agent was utilizing the tools in 76.3% of the cases. Despite the high result for utilizing the tools, in 132 cases, the Security specialist agent missed or misclassified vulnerability in 25.76% of cases.

In this context, another accuracy was introduced, which is a proportion of correctly predicted outcomes out of the total number of cases. And it is calculated as the complement of the "Miss Rate True".

$$Accuracy\ =\ 100\%\ -\ Miss\ Rate\ True\ (\%)$$

In the case above the accuracy would be calculated as $100\ -\ 25.76\ =\ 74.24\%$.

This means that by using the tools, agents' accuracy of correctly identifying vulnerability is 74.24%.

When the tools were not used, the Security specialist agent missed vulnerabilities only in 9.76% cases, making his accuracy 90.26%. This results shows that out of 173 smart contracts, using the tools 132 times proved to be more inefficient with 74.24% of accuracy rather than not using the tools 41 times and yielding 90.26% accuracy.

The Auditor agent utilized the tool 78 times and would not utilize the report 95 times of all cases. This resulted in overall 54.91% of tool usage which is almost half of the time.
However when the tools were used the chance of missing the vulnerability was 32.05% making it 68.95% accurate. When the tools were not used the Auditor agent would misclassify or miss the vulnerability in 14.74% making it 85.26% accurate.

As the Developer agent had no access to the tools his performance is based purely on missing rate of vulnerabilities where reports were not used. This would lead to 100% of not using the tools. As such this metric is acknowledged as biased in this work. However for the general purpose the performance was still calculated for the Developer agent, but denoted with * sign in the table to show an incomplete picture. Following the same calculation steps as previously, the results show that Developer agent would be accurate in correctly identifying vulnerability in 77.51% of all cases where tools are not used.

The results of performance for each of the agents, their tool usage and calculations are presented in Table 10.

Table 10: Tool usage and Accuracy

	Security agent	**Auditor agent**	**Developer agent**
Tool Used (%)	76.3%	54.91%	0%*
Accuracy with tool	74.24%	68.95%	n\a*
Accuracy without tool	90.26%	85.26%	77.51%*

Based on these results it seems that agentic performance is better when the tools are not used. In all cases, even the Developers case which is considered biased, the accuracy is higher where the agents decided not to use the reports. This leads to the belief that the choice of the tools has utmost importance, if the issue does not have a high degree of precision or context-awareness of the vulnerability it may introduce informational noise and degrade the detection process, rather than providing additional value to the agentic general knowledge. This is suggested by the results when comparing vulnerability miss rate when tools were used for Security agent (25.76%) and Auditor (32.05%).

In addition, these findings show that the tool usage itself is not consistent, as some agents decided to use it more and some less. This potentially could be affected by the agent's prompt configuration and role constraints.

4.3.3 Slither performance

Processing with Slither was straightforward. The first iteration of reports included the most information available by the tool, but Slither documentation noted that it is possible to configure the output reports in a way that would drop unnecessary information on demand.
The slither ran locally with additional flags in the command line. These flags were used to generate reports without optimizational, informational and low criticality checks.

Slither tool has the ability to process smart contracts in a batch. However when processing the old Solidity compiler version Slither would throw an error message in the console and all previous reports would not be generated. This led the research to process each contract one by one, switching between older versions for Slither to be able to process old smart contracts.
In one of the instances when the old version was switched to, the slither would still throw an error message for the contract. The reason behind was Solidity language function depreciation. Rewriting the old smart contract to up to date syntax required manual intervention and was considered breaking the integrity of the experiments. Hence it was decided to mark such contracts as False Positives. In this experiment, there were exactly four cases of such contracts that could not be processed. Two of them in Reentrancy classification, one in Unchecked Low

Level Calls and one in Access control. All four cases were marked as False Positive when compared against ground truth. The performance of Slither and other components on macro level is reported in the comparison section.

The performance measures of Slither are presented in Table 11, where the tool could not identify Front Running vulnerability class. However the tool has a high chance of identifying Reentrancy, despite instances where it could not process smart contracts. And Unchecked Low Level Calls with Time Manipulation vulnerability classes are among the classes, where Slither shows high performance, based on the presented data. In addition, Slither could not also identify the Short Address vulnerability that was excluded from the performance evaluation tables.

Table11: Slither macro average performance metrics per class

Vulnerability class	**Accuracy**	**Precision**	**Recall**	**F1**
Access Control	0.58	0.92	0.61	0.73
Arithmetic	0.07	1.00	0.07	0.13
Bad Randomness	0.27	0.5	0.38	0.43
Denial Of Service	0.11	0.25	0.17	0.2
Front Running	0	0	0	0
Other	0.20	0.33	0.33	0.33
Reentrancy	0.94	1	0.94	0.97
Time Manipulation	1	1	1	1
Unchecked Low Level Calls	0.98	1	0.98	0.99
Benign	0.87	0	0	0

With calculating the performance per classification, macro-average and micro-average metrics were again calculated, as per each component of the experiment. In the micro-average Accuracy metric, again, is not applicable to the same reasons stated previously, due to not being balanced. The results for micro-average are represented on Table 12 and macro-average on Table 13 accordingly.

Table 12: Slither Micro average metrics

Micro-Average	
Accuracy	0.699
Precision	0.879
Recall	0.713
F1	0.788

From the tables the performance of the Slither is higher on the micro-average which treats every single instance in the dataset with equal weight. In this case vulnerabilities with more examples have a higher influence on results than other classes.

Table 13: Slither Macro average metrics

Macro-Average	
Accuracy	0.415
Precision	0.600
Recall	0.447
F1	0.478

For the macro-average, when each class has equal weight, the performance shows Slither has a low Recall metric, meaning that the tool misses vulnerabilities in more than half of the cases.

4.3.4 Baseline performance

In this experiment, the baseline model was not given any specific abilities or tools to use, nor was it subjected to fine-tuning training or few-shots examples. The model had to decide on the vulnerability type based on its general knowledge alone. The only information provided was the prompt and SmartBugs vulnerability class mapping, the same mapping used for the multi-agent framework. This was done to remove potential hallucination on vulnerability class. The prompt consisted of simple instructions of taking a role of a baseline and by using knowledge that the

model has, to decide if smart contract was vulnerable YES/NO/CLEAN, with what class according to the mapping.

As seen in Table 14, the baseline could not identify Other classes, returning 0% for the measurements accordingly. However, the model's general knowledge allowed the baseline to exceed in other aspects. These aspects resulted in a high chance identification of Arithmetic, Denial of service, Reentrancy and Unchecked Low Level Calls class vulnerabilities. As no exception to other components of the experiment, the baseline was not able to correctly identify the Short Address vulnerability class which was excluded from the performance evaluation tables. The performance of baseline on macro level, with all the experiment components performance is reported in the comparison section.

Table 14: Baseline macro average performance metrics per class

Vulnerability class	**Accuracy**	**Precision**	**Recall**	**F1**
Access Control	0.55	0.71	0.67	0.69
Arithmetic	1	1	1	1
Bad Randomness	0.18	0.4	0.25	0.31
Denial Of Service	1	1	1	1
Front Running	0.2	0.5	0.25	0.33
Other	0	0	0	0
Reentrancy	0.94	0.97	0.97	0.97
Time Manipulation	0.11	0.2	0.2	0.2
Unchecked Low Level Calls	0.96	0.98	0.98	0.98
Benign	1	0	0	0

Table 15: Baseline Micro average metrics

Micro-Average	
Accuracy	0.771
Precision	0.861
Recall	0.825
F1	0.843

Table 16: Baseline Macro average metrics

Macro-Average	
Accuracy	0.491
Precision	0.575
Recall	0.532
F1	0.548

As previously seen the performance trend is similar to each component where micro-average performance, that gives every single instance in the dataset equal weight, is higher than the macro-average performance where each class has equal weight.

4.3.5 Comparison of performances

Conducted experiment showed that when checking the performance per classification Slither has achieved the performance above 80% in two classes. It has high chances to identify the vulnerability correctly and it has a high chance not to miss vulnerability in these classes. Baseline achieved higher performance in five classes, scoring above 80% where multi-agent achieved higher performance, above 80%, only in two classes. However, when the performance metrics for macro average were calculated the picture shifted. Table 17 shows the macro-average performance of multi-agent is higher in terms of Precision, Recall and F1, and it is more accurate than Baseline and Slither. Baseline has better Recall and F1 metrics than Slither, but less precise. Slither is lacking in performance for all the metrics apart from precision when compared to the baseline model.

Table 17: Comparison of macro-average metrics for components

Macro-average metrics	Slither	Baseline	Multi-agent Forced	Multi-agent Selective
Accuracy	0.415	0.491	**0.568**	0.537
Precision	0.6	0.575	**0.656**	0.612
Recall	0.447	0.532	**0.656**	0.612
F1	0.478	0.548	**0.656**	0.612

The difference is exactly due to the average performance. Previously it was stated that multi-agent had a high succession rate only with two classes, where Slither had four, but overall statistics, which are affected by the low success of other classes, are combined into median performance across all vulnerability classes. Additionally, Slither could not process two smart contracts in the Reentrancy vulnerability class, but as the succession rate with it is or above 94% across measurements the macro average performance should not be affected much.

The results also showed the ability of the Baseline to be used in vulnerability detection as a stand alone tool. Although it had no tools to utilize, it proved to be capable by outperforming Slither in Accuracy, Recall and F1, and also outperforming multi-agent in Accuracy on a macro average level.

Micro-average metrics shown in Table 18, however, show the performance where each smart contract instance of the dataset is treated equally. It should be again noted that the dataset is imbalanced, and each classification is represented by an unequal amount of vulnerable smart contract examples.

Table 18: Comparison of micro-average metrics for components

Micro-average metrics	Slither	Baseline	Multi-agent Forced	Multi-agent Selective
Accuracy	0.699	**0.771**	0.692	0.639
Precision	**0.879**	0.861	0.772	0.730
Recall	0.713	**0.825**	0.804	0.755
F1	0.788	**0.843**	0.788	0.742

An interesting find in this research showed underperformance of all the experiment components in relation to the Short Address vulnerability class. Short Address vulnerability is the attack that occurs when an adversary manipulates the data sent during a transaction, tricking the smart contract into reading more data than was sent. However such low performance can be attributed to the fact that the sample amount in the SmartBugs dataset was limited, where only 1 curated contract contained in the dataset.

The results also highlight that multi-agentic approach with enforced tool usage shows slight but better performance when compared to selective usage approach. Although it is only a few percent increase in performance, it still indicates that potentially the tool usage affects the decision-making in a more positive way, providing the agent a structured and external validation or removing the uncertainty.

These results show an interesting dynamic, because the results also show a contrast that emerges when comparing a multi-agent with a selective approach. In such selective setup agents tend to have better performance when tools are not used. In this perspective it indicates that tool usage is not beneficial and may introduce unnecessary complexity or noise into the decision-making process.

Such observation raises questions related to the internal mechanisms that dictate agentic behavior. In particular, it questions the effectiveness of tool usage which may depend on different factors such as task complexity, agent’s confidence or reasoning, and the quality or relevance of the tool output. Also the percentage of the tool usage could be affected by the prompt

configuration of the research, in both cases for Security specialist and Audit agents the prompt explicitly noted that they both have access to the tools that may or may not be utilized. Although the prompt for both agents had similar wording the difference between tool usage is still noticeable. Security specialist used the tools in 76.3% cases and Audit agent used the tools in 54.91% cases. This makes a difference of 21.39% between both agents. As security agent is focused on overall blockchain and vulnerability patterns and mapping it leads to a higher tool usage. Where the Auditor agent focusing on smart contract review and logic neglected the tool more to potentially avoid informational noise. However, forcing the tool ensures that high-level critical vulnerabilities checks are never missed by the agent which may have led to increased performance metrics.

It is possible that in more simple cases, direct model reasoning, based on general knowledge and trained data, is sufficient. Whereas tool invocation may disrupt the reasoning, by potentially introducing misleading complexity and affecting the agent.

These findings highlight the need for deeper research in the related field. And answer the question on how agents decide when to use tools and how tool outputs are integrated into their reasoning processes. Alongside how prompt configuration or the roles assignation to the agents affects the confidence of decision-making process and therefore the tools usage.

Understanding this internal decision-making dynamic remains an open research question and represents a potential direction for future work. At the same time the choice of the tools is also important. During the experiment, it was understood that the tools should not only be up to date, but also backwards compatible with older versions of smart contracts. This requires the tool to be well maintained and further developed.

5 Limitations

The biggest limitation encountered was not associated with the multi-agent framework, but with the lack of curated smart contracts of the dataset. Another observation is a potential sensitivity to prompt configuration and role constraints. The reason for this is potentially the results of the experiment that highlights that selective and forced differ between usage of the tool percentage wise, however the reason behind it stays underexplored.

5.1 The Dataset

This research was built on the base of SmartBugs dataset, which consists of 10 different vulnerability classes. Each vulnerability class has a different amount of curated smart contracts. The limitation here is the amount and unbalanced dataset, as an example Short Address class has only 1 curated smart contract where Unchecked Low Level Call has 52 smart contracts.
As the dataset consists of 143 smart contracts of vulnerable contracts and 30 benign smart contracts, ranging in each vulnerability class, which shows that a bigger dataset would be more beneficial for measuring performance of the future research. The SmartBugs dataset, consisting of 143 smart contracts, is less suited for more specific approaches where LLM should be taught based on the broad dataset, such as fine-tuning, that requires the dataset to be separated into three subsets - training, validation and test set. This type of dataset would not be suitable for fine-tuning, unless used for a specific vulnerability class. Also, at the time of the writing, fine-tuning was not available for Claude models. However for some approaches, such as one-shot or few-shots this dataset can be proved sufficient, as it will cover most of the vulnerabilities apart from Short Address class.

As the dataset included the benign data, one thing to point out is that any of the tested vulnerability classes, including clean smart contracts, could be in the general knowledge of the model.

5.2 The Mapping

In any research, to get accurate results all the dataset points must be coherent. In this work SmartBugs dataset was used with all curated smart contracts. Each smart contract is annotated with vulnerability class and comment in the code section where exactly is the vulnerability

happening. However, when compared to static analysis tool, such as Slither, the underlying working logic of the tool is based on vulnerability checks, such as “unchecked-lowlevel” or “reentrancy-eth”. This may create a discrepancy between vulnerability classes and vulnerability checks. The mapping of Slither checks and SmarBugs classes were done in accordance with two different documentation references. Incorrect mapping will lead to incorrect performance results as all components of the experiment were based on correctness of this mapping. The experiment was done with the assumption that the mapping was done correctly.

5.3 The orchestration space and agentic tools

As the Large Language Model evolves, tools for their orchestration are also evolving, but in case of this research, LLM and blockchain ecosystem combined, would require more specific tools. By utilizing 3rd party service for multi-agent orchestration, it became apparent that vulnerability detection tools are not integrated into the orchestration space, or the tools are not commonly known yet. Due to this fact the agents built on the service were given the tool’s reports for decision making, but they were not provided with the tools themselves. In this regard, the research acknowledges this limitation as lack of tool integration, however a survey if the tool was called by the agents was still conducted.

5.4 The Slither

Slither is a static analysis tool for vulnerability detection in smart contract, the underline logic of the tool requires for smart contract to be compilable. This research has come across a few instances where Slither could not process smart contracts, one from Unchecked Low Level Calls and two from Reentrancy classes. The results from Slither in these cases were empty json reports with compiler error related to the smart contract old compiler version of Solidity language. After changing the version, the issue persisted, it was due to the fact that code syntax used during the older version were now deprecated. If this syntax would be fixed manually for the research, that would affect the result in a slight percentage, hence it was decided to treat such instances as False Positives against the ground-truth.

6 Discussion

This section synthesizes the experimental results and interprets the performance of the evaluated components which consists of Slither, a baseline Claude model, and the multi-agent Claude framework. Multi-agent framework was additionally analysed separately as a tool-augmented approach with selective and force tool usage. All these components are evaluated within the context of smart contract vulnerability detection.

6.1 Synthesis of results and main findings

The result in this research shows a variation in detection performance across vulnerability classes. One potential explanation is dataset imbalance, Table 19 shows the amount of examples in the dataset per classification and Accuracy metric per each component. A clear correlation is seen where underrepresented vulnerability classes such as Other, Front Running, Time Manipulation and Short Address contain fewer examples than other classes. In contrast, classes like Reentrancy, Arithmetic and Access Control with sufficient examples provide more consistency, which may increase detection rate.

Table 19: Smart contracts amount compared to Accuracy of all components

	Smart contracts	**Slither**	**Baseline**	**Multi-agent Forced**	**Multi-agent selective**
Access Control	18	0.58	0.52	0.89	0.89
Arithmetic	15	0.07	1.00	1.00	1.00
Bad Randomness	8	0.27	0.18	0.78	0.78
Denial Of Service	6	0.11	1.00	0.71	0.71
Front Running	4	0	0.20	0.33	0.14
Other	3	0.20	0	0.20	0.20
Reentrancy	31	0.94	0.94	1.00	1.00
Short Address	1	0	0	0.00	0.00
Time Manipulation	5	1.00	0.11	0.25	0.25
Unchecked Low Level Calls	52	0.98	0.96	0.51	0.39
Benign	30	0.87	1	0.8	0.83

6.1.1 Difficulty of vulnerability classes detection

Another factor may be the complexity of smart contract logic. Vulnerabilities such as Time Manipulation or Front Running may depend on context, in such cases the order of transaction execution on blockchain is important. Such vulnerabilities are less explicit in the code and more difficult to detect. This may lead to agentic inconsistency and increase misclassification by assigning into broader categories such as Other. This is especially relevant in bigger reports which generate more noise as these tools may fail to capture vulnerabilities which are not based on patterns.

Finally, it is possible that a necessity to assign a single vulnerability class per contract may contribute to inaccurate classification. Some vulnerability classes may share similar patterns, making strict categorization challenging for multi-agentic architecture. This suggests that lower performance in certain classes is a combination of dataset characteristics, vulnerability complexity, and architectural constraints of the multi-agent framework.

6.1.2 Components performance against Slither

Overall the results presented in Tables and Figures in the “Results and analysis” section show clear differences between the evaluated approaches in terms of precision, recall, and F1-score.
On a macro-average level the multi-agent framework outperforms Slither in all performance metrics. Multi-agent Forced recall (65%) and multi-agent selective recall (61.2%) are higher compared to the Slither (44.7%), indicating a stronger ability to detect existing vulnerabilities both on macro and micro average levels.

However baseline showed better performance on micro-average level where each smart contract is given the same weight across the imbalance dataset. This could be explained by the fact that the baseline, when processed smart contract was not affected by the noise of the tools reports.

These results suggest that multi-agent frameworks improve vulnerability detection through complementary reasoning, where agents with different roles and tools contribute their perspective to the matter.

6.1.3 Mutli-agent force and multi-agent selective insight

The insight for selective and forced tool usage highlighted that tool integration is not inherently beneficial, but depends on the context and how the tool is used within the agentic workflow. The Security agent missed vulnerabilities in 25.76% of cases when using the tool, compared to 9.76% when not using it, while the Auditor missed 32.05% with the tool and 14.74% without it. This suggests that tool choice can negatively affect performance. However, the factors influencing tool usage itself remain unclear, particularly given the fact that the tool was utilized more frequently by the Security Agent in 21.39% cases than by the Auditor. This potentially suggests that the role specifications assigned to each of the agents could affect internal decision-making processes.

6.2 Answering Research Questions

1. How effective is a multi-agent framework compared to traditional static analysis tools for smart contract vulnerability detection?

The results of this research shows that a multi-agent framework proved to be an effective and competitive approach that outperforms Slither static analysis tool.

On a micro-average level multi-agent framework surpasses Slither in Recall metric, indicating higher chance to identify vulnerabilities that traditional tools may miss. Slither, however, showed higher Precision. This could be attributed to the fact that in the scope of this research, the output for Slither was configured in a way to drop optimizational, informational and low criticality checks. This results in reporting only high-confidence vulnerabilities, which could potentially increase the Precision in a multi class classification. Alternatively Slither operates on pre-defined vulnerability checks, where multi-agent framework operate on internal knowledge and internal reasoning, which can override report results provided with the input, leading to increased false positives that would lower the precision.

On a macro-average level, where each vulnerability class is assigned equal weight in an imbalanced dataset multi-agent framework, both selective and forced, demonstrated higher performance across all evaluation metrics (Precision Accuracy, Recall and F1-score), when compared to Slither tool. Forced multi-agent framework achieved Precision, Recall and F1-score resulting in 65.6% and Accuracy 56.8% followed by selective multi-agent with Precision, Recall and F1-score resulting in 61.2% and Accuracy 53.7%. Both frameworks outperform Slither's Precision 60%, Recall 44.7%, Accuracy 41.5% and F1-score 47.8%. The results comparison can be seen on Table 17.

2. How effectively does a role-based multi-agent framework detect vulnerabilities across standard evaluation metrics, including recall, precision, and F1-score?

The evaluation of the role-based multi-agent framework and the baseline in detecting vulnerabilities was measured using performance metrics, such as accuracy, recall, precision, and F1-score. The results demonstrate that the multi-agent framework, with forced approach, achieves the strongest performance, with accuracy 56.8% and recall, precision and F1-score

reaching 65.6%. This approach outperforms both the selective multi-agent and the baseline model. The selective approach shows moderate performance with accuracy 53.7% and recall, precision and F1-score reaching 61.2%, while the baseline model performs with lower accuracy 49.1%, recall 53.2%, precision 57.5% and F1-score 54.8%.

These findings show that the multi-agent approach improves the detection of vulnerabilities detection. However, similar values of precision and recall suggest that false positives remain a relevant concern as performance varies across vulnerability classes. Higher detection rate is observed in classes that are well-represented and structurally explicit and reduced detection rate is observed in the classes with less examples or in context-dependent vulnerabilities.

Overall, the role-based multi-agent framework can be considered more effective than the baseline model when evaluated across all standard metrics, when each vulnerability class is given the same weight. Although the performance may be influenced by dataset imbalance and the complexity of certain vulnerability types.

3. How does forced tool usage compare to selective tool usage in terms of agentic performance?
The comparison of the two approaches demonstrates that forced tool usage achieves higher performance in vulnerability detection tasks. This suggests integrating external tools into the agentic workflow can improve detection capability by providing additional information that can support agentic reasoning.

However, an interesting dynamic emerges when analyzing the selective multi-agent approach. The results show that agents are more accurate in cases when tools are not used. Specifically, the Security agent missed vulnerabilities in 25.76% of cases when using the tool, compared to 9.76% when not using them. Similarly, the Auditor agent exhibited a higher miss rate when using the tool 32.05% compared to not using it with 14.74% of missing rate. This suggests that tool integration does not inherently improve performance and may introduce noise during reasoning processes.

Furthermore, differences in tool usage frequency between agents indicate that role-specific behavior influences decision-making. The Security agent utilized tools 21.39% more frequently than the Auditor, which may reflect differences in role definitions and internal reasoning. This suggests that the effectiveness of tool usage is not only dependent on the tool itself, but is a combination of role configuration and the tool application within the agentic workflow.

Overall, while forced tool usage provides more stable and higher performance, selective tool usage reveals that tool effectiveness is context-dependent. This highlights an important area for future research in optimizing tool integration with multi-agent framework.

6.3 Areas for future research

After conducting an experiment this research highlights several challenges and opportunities for further research in the field of multi-agent vulnerability detection framework.

One of the key points is the relevance of external tools used within agentic systems. The results indicate that integrating static analysis tools can affect detection performance. However, outdated or limited tools may not give any additional information to the agents beyond the model's general knowledge. Alternatively, tools that generate reports with a high volume of informational and low criticality information may negatively impact the performance of an agentic workframe. Potential opportunity for the future research could be an investigation of selection mechanisms, where agents evaluate how useful the tool is before integrating its output into the decision-making process. This could be introduced as a confidence score where the agents could assign a trust score to the tool output, ultimately deciding whether tool usage is beneficial against internal reasoning and cross verify between other agents. Another direction is context aware of the tool triggering or the tool's noise filtering also based on the confidence score of the agent, this would allow the tool to utilize the tool in complex cases where tools are justified rather than being used uniformly.

Another opportunity for the research is to understand how prompt configuration or the roles assignment, or their constraints, to the agents affects the confidence of the output. In this

research, agents were assigned predefined roles with fixed instructions. While this ensures reproducibility of the experiment, it may restrict the reasoning capabilities and potentially tool usage, which in the end could affect the performance.

A further direction could explore individual vulnerability classes. Each vulnerability class may require different detection characteristics. Future research could explore and understand which vulnerabilities are consistently misclassified and the reason behind it. As an example the underlined misclassification of Short Address vulnerability class by all components, in this experiment was acknowledged due to dataset limitations. However, another reason may be the agent's internal reasoning process, where the vulnerability is interpreted as a different class. While this may be considered a false positive in terms of classification against ground-truth, it still indicates that the components identified a potential security issue. To address these challenges, future research could explore the use of specialized agents, where each agent is assigned a specific vulnerability class with configured role and prompt for the class accordingly. The prompt configuration should include patterns of the vulnerability class if applicable. Underrepresented vulnerability classes could benefit from synthetic data generation. This would resolve the issue with an imbalanced dataset and increase pattern recognition. Another promising direction is the implementation of class-specific toolset, where appropriate tools are mapped to particular vulnerability types and their outputs are evaluated using confidence scoring. Such an approach could potentially increase the detection rate across common and rare vulnerability classes.

Improving the multi-agentic framework remains a challenge. Future work could focus more on agentic behavior by implementing explainable steps behind agentic reasoning. This would improve transparency and potential insights of agents' decisions on the tool usage or the confidence of the tool usage. In such cases a confidence scoring could be introduced, where final decisions are weighted based on the certainty of each agent. Furthermore, future research could explore a dynamic role adaptation, where the role or behavior is adjusted depending on the complexity of the input or when consensus between the agents is not met. Such approaches could potentially improve reliability of the multi-agent framework.

7 Conclusions

The main goal of this research was to evaluate the performance of a role-based multi-agent system built on Claude Sonnet 4.5. And to see if it can improve the effectiveness of Ethereum smart contract vulnerability detection compared to a traditional static analysis tool and a baseline Large Language Model using the same Claude 4.5 model. This work utilized the publicly available SmartBugs dataset and did not rely on fine-tuning or few-shots examples. Additionally, the study compared the performance of forced versus selective tool usage within a tool-augmented multi-agent framework. To the best of the author's knowledge, such a direct comparison within a reproducible experiment setup has not been previously explored.

The results of the experiments show that the multi-agent system is capable of achieving competitive performance across evaluation metrics, including precision, recall, and F1-score. The multi-agent framework, both forced and selective tool usage, showed improved recall and precision compared to the static analysis tool. This indicates that such an approach has a feasible ability to detect existing vulnerabilities. Forced tool usage, when compared to selective tool usage, has slightly higher performance metrics.

While multi-agent framework show promise, its cost-effectiveness and scalability require further investigation, particularly around token usage and latency. Although token usage was measured during the experiments, the results are limited to the SmartBugs [10] dataset and the specific experimental setup of this study, and therefore may not reflect the general deployment scenarios. On the other hand, while the baseline model demonstrates competitive performance across several evaluation metrics, its application may be limited. As an example, when processing a high volume of smart contracts, daily usage quotas can be quickly exhausted, which would require access to paid tiers for further operation. In this context, the multi-agent framework, despite its higher per-instance computational cost, may offer a more structured and potentially scalable approach when integrated into controlled workflows.

Rather than replacing traditional tools, the multi-agent framework can be complemented by them. Results in this work show that mutual collaboration with correct setup can improve detection rate and identification of vulnerabilities.

Based on the findings, it can be concluded that multi-agent architectures do show a promising direction for smart contract vulnerability detection. These systems are effective, have higher recall and provide broad vulnerability coverage. However, dataset imbalance, data size and refined prompt with role assignment could potentially improve the systems before it can be fully relied on in smart contract security and audit applications.

Future research could potentially explore how agents decide when to use tools and how tool outputs are integrated into their reasoning processes. Alongside how prompt configuration or the roles assignation to the agents affects the confidence of decision-making process and therefore the tools usage. Understanding this internal decision-making dynamic remains underexplored.

In conclusion, this research demonstrates that a multi-agent framework, when properly configured with refined prompt and accurate tools, can improve smart contract vulnerability detection, while also showing the limitations and pointing out the areas for future research in this emerging field.

References

[1] DeFiLlama. "Chains." DeFiLlama. https://defillama.com/chains (accessed: Apr. 18, 2026)

[2] N. Atzei, M. Bartoletti, and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts SoK," in Proc. 6th Int. Conf. Principles of Security and Trust, vol. 10204, Berlin, Heidelberg: Springer-Verlag, 2017, pp. 164–186.

[3] M. F. Andrijasa, S. A. Ismail, N. Ahmad, and O. M. Yusop, "Enhancing smart contract security through multi-agent deep reinforcement learning fuzzing: A survey of approaches and techniques," Int. J. Adv. Comput. Sci. Appl. (IJACSA), vol. 15, no. 5, 2024.

[4] Chainalysis. "Crypto Hacking Stolen Funds 2026." Chainalysis. https://www.chainalysis.com/blog/crypto-hacking-stolen-funds-2026/ (accessed: Apr. 07, 2026).

[5] S. So, S. Hong, and H. Oh, "SmarTest: Effectively hunting vulnerable transaction sequences in smart contracts through language model-guided symbolic execution," in Proc. 30th USENIX Security Symp., Virtual, 2021.

[6] H. Wu, Z. Zhang, S. Wang, Y. Lei, B. Lin, Y. Qin, H. Zhang, and X. Mao, "Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques," in Proc. IEEE 32nd Int. Symp. Softw. Reliab. Eng. (ISSRE), Wuhan, China, 2021.

[7] W. Ma, D. Wu, Y. Sun, T. Wang, S. Liu, J. Zhang, Y. Xue, and Y. Liu, "Combining fine-tuning and LLM-based agents for intuitive smart contract auditing with justifications," arXiv preprint, 2024

[8] Anthropic. "Claude." Anthropic. https://www.anthropic.com/ (accessed: Dec. 08, 2025).

[9] n8n. "n8n." n8n. https://n8n.io/ (accessed: Dec. 08, 2025).

[10] SmartBugs. "SmartBugs Curated Dataset." GitHub. https://github.com/smartbugs/smartbugs-curated (accessed: Dec. 12, 2025).

[11] OpenZeppelin. "OpenZeppelin Contracts." GitHub. https://github.com/openzeppelin/openzeppelin-contracts (accessed: Dec. 12, 2025).

[12] Enzyme Finance. "Oyente." GitHub. https://github.com/enzymefinance/oyente (accessed: Dec. 10, 2025).

[13] ConsenSys Diligence. "Mythril." GitHub. https://github.com/ConsenSysDiligence/mythril (accessed: Dec. 10, 2025).

[14] Crytic. "Slither." GitHub. https://github.com/crytic/slither (accessed: Dec. 10, 2025).

[15] Trail of Bits. "Manticore." GitHub. https://github.com/trailofbits/manticore/ (accessed: Dec. 10, 2025).

[16] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in Proc. 2016 ACM SIGSAC Conf. Comput. Commun. Security (CCS '16), Vienna, Austria, Oct. 2016, pp. 254–269.

[17] Sharma, Nipun & Sharma, Swati. (2022). A Survey of Mythril, A Smart Contract Security Analysis Tool for EVM Bytecode. 13. 51003-51010.

[18] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 Ethereum smart contracts," in Proc. 42nd Int. Conf. Softw. Eng. (ICSE), Seoul, Republic of Korea, 2020, pp. 530-541.

[19] J. Feist, G. Grieco, and A. Groce, "Slither: A Static Analysis Framework For Smart Contracts," in Proc. 2nd British Int. Conf. on Next Generation Software Engineering (NGSE), 2019.

[20] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart

Contracts," in Proc. 34th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE), San Diego, CA, USA, 2019.

[21] C. Chen, J. Su, J. Chen, Y. Wang, T. Bi, J. Yu, Y. Wang, X. Lin, T. Chen, and Z. Zheng, "When ChatGPT meets smart contract vulnerability detection: How far are we?," ACM Trans. Softw. Eng. Methodol., vol. 33, no. 8, 2024.

[22] LLM-SmartAudit , Z. Wei, J. Sun, Y. Sun, Y. Liu, D. Wu, Z. Zhang, X. Zhang, M. Li, Y. Liu, C. Li, M. Wan, J. Dong, and L. Zhu, "Advanced smart contract vulnerability detection via LLM-powered multi-agent systems," 2025.

[23] T.-T.-H. Le, J. Kim, S. Lee, and H. Kim, "Robust Vulnerability Detection in Solidity-Based Ethereum Smart Contracts Using Fine-Tuned Transformer Encoder Models," *IEEE Access*, vol. 12, pp. 154700–154717, 2024.

[24] SmartBugs. "SmartBugs Wild Dataset." GitHub. https://github.com/smartbugs/smartbugs-wild (accessed: Dec. 10, 2025).

[25] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "SmartBugs: A Framework to Analyze Solidity Smart Contracts," INESC-ID and IST, University of Lisbon, Portugal, and KTH Royal Institute of Technology, Sweden, 2024.

[26] S. Y. Chavhan, S. Kumar, and A. Karkare, "ScrawlD: A Dataset of Real World Ethereum Smart Contracts Labelled with Vulnerabilities," Indian Institute of Technology Kanpur, Kanpur, India, 2024.

[27] LLM-BSCVM, Y. Jin, C. Li, P. Fan, P. Liu, X. Li, C. Liu, and W. Qiu, "LLM-BSCVM: An LLM-based blockchain smart contract vulnerability management framework," Guangxi Normal University, China, May 2025.

[28] SPEAR, A. Mallick, I. Chebolu, and H. Rana, "SPEAR: An engineering case study of multi-agent coordination for smart contract auditing," Center for Development of Advanced Computing, Hyderabad, India. Available: https://www.cdac.in, February 2026.

[29] Smartify, Karanjai, R., Blackshear, S., Xu, L. and Shi, W., 2025. A multi-agent framework for automated vulnerability detection and repair in solidity and move smart contracts. arXiv e-prints, pp.arXiv-2502.

[30] Crytic. "Not-So-Smart-Contracts." GitHub. https://github.com/crytic/not-so-smart-contracts [Accessed: Mar. 10, 2026].

[31] Code4rena. "Code4rena." Code4rena. https://code4rena.com/ (accessed: Dec. 08, 2025).

[32] JetBrains. "PyCharm." JetBrains. https://www.jetbrains.com/ (accessed: Dec. 08, 2025).

[33] SmartBugs. "Vulnerabilities Mapping." GitHub. https://github.com/smartbugs/smartbugs/wiki/Vulnerabilities-mapping (accessed: Dec. 12, 2025).

[34] Crytic. "Detector Documentation." GitHub. https://github.com/crytic/slither/wiki/Detector-Documentation (accessed: Dec. 12, 2025).

[35] Ethereum. "Token Standards." Ethereum. https://ethereum.org/developers/docs/standards/tokens/ (accessed: Apr. 29, 2025).

[36] SolidityScan. "SolidityScan." SolidityScan. https://solidityscan.com/ (accessed: Dec. 10, 2025).

[37] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," Information Processing & Management, vol. 45, no. 4, pp. 427–437, 2009.

Appendix 1

Slither and SmartBugs vulnerability mapping

arbitrary-send-eth - Access control
constable-states - Other
divide-before-multiply - Arithmetic
unchecked-lowlevel - Unchecked Low Level Calls
calls-loop - Denial of Service
reentrancy-benign - Reentrancy
timestamp- Time Manipulation
weak-prng- Bad Randomness
controlled-array-length- Access control
reentrancy-eth - Reentrancy
constant-function-asm- Other
erc20-interface - Other
events-access - Other
events-maths - Other
external-function - Other
external-function - Other
controlled-delegatecall - Access control
missing-zero-check - Access control
locked-ether - Denial of Service
incorrect-equality - Other
reentrancy-events - Reentrancy
incorrect-modifier - Other
unchecked-send - Unchecked Low Level Calls
shadowing-builtin - Other
shadowing-local - Other
shadowing-state - Other
suicidal - Access control

tx-origin - Access control

unchecked-transfer - Unchecked Low Level Calls

tautology - Other

return-bomb - Denial of Service

uninitialized-local - Other

uninitialized-state - Other

uninitialized-storage - Other

unused-return - Other

Appendix 2

Agentic output

CRITICAL INSTRUCTIONS:
- Output ONLY valid JSON
- Keep description under 50 words
- Use exact category names listed above
- If multiple vulnerabilities exist, report the most critical one
- No explanations, no additional text and no formatting outside the JSON.

Provide ONLY the JSON response below.

Analyze the smart contract for vulnerabilities and respond with EXACTLY this JSON structure:

{
"vulnerability": "Yes/No",
"type": "vulnerability category from:
Access control, Arithmetic,
Front Running, Reentrancy,
Short Addresses,
Time Manipulation,
Unchecked Low Level Calls,
Denial of Service,
Bad Randomness,
Other,
Clean",
"2_sentence_description": "Brief description of the vulnerability location and impact, or 'Contract is clean' if no vulnerability found",
"confidence": "High/Medium/Low"

Appendix 3

Smart Contract Auditor Input

You are a Smart Contract Auditor responsible for comprehensive contract review and final decision.
You have access to the following tool: - get_solidityscan_report: Returns a detailed static analysis report from Solidityscan.
Decide whether to use this tool based on your needs.
Set \"tool_used\" to true if you want to use the report, false if you don't need it.
Analyze the smart contract for vulnerabilities and respond with EXACTLY this JSON structure:

```
{
    \"tool_used\": true/false,
    \"vulnerability\": \"Yes/No\",
    \"type\": \"Access Control | Arithmetic | Front Running | Reentrancy | Short Addresses | Time Manipulation | Unchecked Low Level Calls | Denial of Service | Bad Randomness | Other | Clean\",
    \"2_sentence_description\": \"Brief description of the vulnerability location and impact, or 'Contract is clean' if no vulnerability found.\",
     \"how_report_helped\": \"One sentence on how the report helped, or 'get_solidityscan_report was not used'.\",
    \"confidence\": \"High | Medium | Low\"
}
```

RULES: - Set \"tool_used\" to true if you called get_solidityscan_report, false if you analyzed without it - If tool was used, factor its findings into your assessment - If multiple vulnerabilities exist, report the most critical one - Keep description under 50 words - Use exact category names listed above - Output ONLY valid JSON

Smart Contract Developer Input

You are an expert Ethereum Smart Contract Developer specializing in Solidity security analysis.
You have no tools available, you can safely set tool_used to false.
CRITICAL: Provide ONLY the JSON response below.
No explanations, no additional text, no formatting outside the JSON.
Analyze the smart contract for vulnerabilities and respond with EXACTLY this JSON structure:
{

\"tool_used\": true/false,

\"vulnerability\": \"Yes/No\",

\"type\": \"Access Control | Arithmetic | Front Running | Reentrancy | Short Addresses | Time Manipulation | Unchecked Low Level Calls | Denial of Service | Bad Randomness | Other | Clean\",

\"2_sentence_description\": \"Brief description of the vulnerability location and impact, or 'Contract is clean' if no vulnerability found\",

\"how_report_helped\":\"1 sentence how report helped, or 'N/A'\

\"confidence\": \"High/Medium/Low\"

}
RULES:

- Output ONLY valid JSON

- Keep description under 50 words
- Use exact category names listed above
- If multiple vulnerabilities exist, report the most critical one
- No markdown, no code blocks, no explanations - ONLY JSON"

Smart Contract Security Specialist Input

You are a Smart Contract Security Specialist with deep expertise in blockchain security.
You have access to the following tool:
- get_slither_report: Returns a detailed static analysis report from the Slither tool.
Decide whether to use this tool based on your needs. Set \"tool_used\" to true if you want to use report, false if you don't need it.

Provide ONLY the JSON response below. No explanations, no additional text, no formatting outside the JSON.
Respond with EXACTLY this JSON structure:

```
{
    \"tool_used\": true/false,
    \"vulnerability\": \"Yes/No\",
    \"type\": \"Access Control | Arithmetic | Front Running | Reentrancy | Short Addresses | Time Manipulation | Unchecked Low Level Calls | Denial of Service | Bad Randomness | Other | Clean\",
    \"2_sentence_description\": \"Brief description of the vulnerability location and impact, or 'Contract is clean' if no vulnerability found\",
    \"how_report_helped\":\"1 sentence how report helped, or 'get_slither_report was not used'\"
    \"confidence\": \"High/Medium/Low\"
}
```

When you check Slither report, Slither checks can be mapped as following:
[Appendix 1 Slither and SmartBugs vulnerability mapping]
RULES: -Set \"tool_used\" to true if you called get_slither_report, false if you analyzed without it
- Output ONLY valid JSON
- Keep description under 50 words
- Use exact category names listed above
- If multiple vulnerabilities exist, report the most critical one
- No markdown, no code blocks, no explanations - JUST JSON

Appendix 4

True/False boolean per classification for selective tool usage

Category	Role	Tool Used (True)	Tool Not Used (False)	Missed (True)	Missed (False)
Reentrancy	Security Specialist	25	6	0	0
	Auditor	7	24	0	0
	Developer	n\a	31	n\a	0
Benign	Security Specialist	16	14	3	1
	Auditor	17	13	5	0
	Developer	n\a	30	n\a	5
Unchecked Low Level Calls	Security Specialist	51	1	22	0
	Auditor	32	20	14	8
	Developer	n\a	52	n\a	20
Access Control	Security Specialist	13	5	1	0
	Auditor	9	9	1	0
	Developer	n\a	18	n\a	1
Arithmetic	Security Specialist	7	8	0	0
	Auditor	2	13	0	0
	Developer	n\a	15	n\a	0
Bad Randomness	Security Specialist	7	1	1	0
	Auditor	5	3	1	0
	Developer	n\a	8	n\a	1
Short Address	Security Specialist	1	0	1	0
	Auditor	1	0	1	0
	Developer	n\a	1	n\a	1
Denial Of Service	Security Specialist	5	1	1	0
	Auditor	3	3	1	0
	Developer	n\a	3	n\a	2

Front Running	Security Specialist	3	1	2	1
	Auditor	1	3	1	2
	Developer	n\a	3	n\a	3
Other	Security Specialist	2	1	2	0
	Auditor	1	2	1	1
	Developer	n\a	3	n\a	2
Time Manipulation	Security Specialist	2	3	1	2
	Auditor	0	5	0	3
	Developer	n\a	5	n\a	3