AuditCopilot: An End-to-End LLM-Based Framework for Automated Vulnerability Detection in Web3

Zhiju Yang[1][0000−0003−2831−7424], Xiaoshuai Zhang[2], and Xin Zhao[1]

[1] Seattle University, Seattle, USA
zhijuyang@seattleu.edu, xzhao1@seattleu.edu
[2] Ocean University of China, Qingdao, China
x.zhang@ouc.edu.cn

Abstract. Security audits for Web3 projects are expensive and time-consuming, and existing static and LLM-based tools often overwhelm auditors with noisy, weakly supported alerts. What auditors need in practice is a workflow that preserves coverage while sharply improving precision and evidence quality. AuditCopilot is an end-to-end, LLM-driven workflow that helps human auditors review and audit Web3 code with fewer false positives. The scanner runs multiple large language models independently and aggregates candidate findings. A four-stage agentic validator then re-proves each candidate through: (i) rules-based invalid filtering, (ii) main reasoning with tool use (local file search, semantic retrieval, and web search), (iii) missing-check analysis for access control and input validation, and (iv) intended-design confirmation to suppress features that match documented or standard behavior. Across 14 real audit projects the scanner produced 642 raw findings; after validation and de-duplication only 45 were reported, and auditors confirmed 33 (73.3% precision; median 2 findings per project). In a head-to-head study on 108 files (181 raw findings), precision rose from 24.6% (single vanilla LLM) to 55.0% (LLM judge voting) and to 87.5% with our agentic validator. The precision-first design intentionally trades some recall for auditor trust and lower triage cost. These results show that combining multi-model scanning with staged, tool-grounded validation can deliver concise, high-confidence findings in real audits while keeping auditors in control.

Keywords: Vulnerability Detection · Web3 · LLM

1 Introduction

Security audits for Web3 systems are costly and slow, while projects usually ship fast, codebases are large, and attack surfaces evolve daily. Traditional static analyzers help, but they miss context and flood auditors with false positives. Recent LLM-based tools show promise, yet single-pass prompts or one-shot verdicts struggle to maintain precision in real audits. What auditors need is a workflow that can scale, focus on what matters, and back every claim with evidence.

We present *AuditCopilot*, an end-to-end framework that uses LLMs to automate the audit pipeline while keeping human auditors in control. It offers two complementary ways to explore a project: a full-project sweep for broad coverage and a targeted pass that prioritizes key files and functions. A checklist dataset (47 categories, covering more than 1,500 checks) drives a systematic review across common and edge-case patterns. Findings do not go straight to the report; instead, a multi-stage validator re-proves each candidate with independent reasoning and tool-grounded tests. It first filters common invalid patterns, then runs a main validation pass, then examines access control and input validation, and finally confirms intended design. Only confirmed issues advance to reporting. This design prioritizes precision and complements human auditing.

This approach matters in real audits where precision is the bottleneck. Invalid findings waste time and erode auditor trust. By making the validator a first-class stage rather than an afterthought, AuditCopilot reduces noise and preserves auditors' attention for high-impact issues. We also compared this validator against vanilla LLMs and LLM-judge voting baselines; it outperformed both by delivering fewer but more reliable findings, raising the valid-to-invalid ratio and improving overall precision. Overall, this paper makes the following contributions:

- **A practical LLM audit framework.** An end-to-end system that blends full-project and key file scanning with a checklist-guided pass, aligned with real audit workflows.
- **Agentic validator for precision.** A multi-stage, tool-assisted validator that filters invalid patterns, re-checks logic, and executes targeted tests before reporting.
- **Structured checklist at scale.** A curated list of **47** categories (**1,500+** checks) that broadens coverage without inflating false positives.
- **Evidence from live audits.** Evaluation on **14** real audit projects. Across projects, scanners produced **642** raw findings; the system reported **45** after validation with an overall precision of **73.3%** (Table 2). In a head-to-head study (108 files, 181 raw findings), precision rose from **24.6%** (vanilla LLM) to **55.0%** (LLM judges) and to **87.5%** with our agentic validator (Table 3).

Our goal is not to replace human auditors, but to cut noise and surface high-confidence issues faster. The validator favors precision by design; some low severity or impractical issues are intentionally filtered. In practice, such coverage gaps can be complemented by traditional tools (e.g., static analyzers), while AuditCopilot focuses on delivering high-confidence and practical security issues that support human auditors in real-world audits.

The rest of the paper is organized as follows. Section 2 reviews the landscape that motivates a precision-first design. Section 3 details the architecture and each component of AuditCopilot. Section 4 evaluates its performance by reporting results from real audit projects. Section 5 discusses limits and future work, and Section 6 concludes.

2 Background and Related Work

2.1 Traditional Vulnerability Detection in Smart Contracts

Traditionally, researchers focused on static analysis, dynamic analysis, and formal verification techniques for smart contract vulnerability detection. Popular static analyzers such as Oyente, Mythril, Securify, Slither, and SmartCheck check for known vulnerability patterns using manually crafted rules [3–7]. These tools are effective at identifying issues like reentrancy and integer overflows, which supports web3 auditing efforts [15]. But they often suffer from high false positive rates and limited flexibility [18]. Rule-based detectors require continuous updates for new vulnerability types and lack deep semantic understanding of code, which leads to missed issues or outdated checks [13, 18]. Dynamic approaches (e.g., fuzzers and runtime monitoring) and formal methods (e.g., Zeus) have likewise been applied [8], but they are constrained by execution path explosion and scalability issues. In summary, traditional methods rely heavily on expert knowledge and predefined patterns, which limits their automation, efficiency, and adaptability considering the fast evolving Web3 attack vectors [13].

2.2 Deep Learning-Based Approaches

Researchers have increasingly explored machine learning, especially deep neural networks, to automatically learn vulnerability patterns from data [13]. Instead of manually crafted rules, these models can infer code characteristics correlated with security flaws. Sequence-based models were among the earlier attempts: for example, Qian et al. [9] trained sequential deep models to detect reentrancy vulnerabilities, showing that neural networks can recognize complex patterns beyond simple heuristics. Other works treat contract bytecode or source code as a sequence of tokens/opcodes and apply RNNs or Transformers for classification of vulnerable contracts or functions.

To incorporate richer code semantics, many approaches construct graph representations of contracts and apply graph neural networks (GNNs). Examples include MANDO (Nguyen et al., 2022) [10], Peculiar (Wu et al., 2021) [11], and VDDL (Jiang et al., 2022) [12], all of which model smart contracts as heterogeneous graphs or dependency graphs. These GNN-based models eliminate the need for explicit rules by learning structural patterns of vulnerabilities. However, later studies pointed out that using any single graph type may lose semantic information, since each graph captures only part of the contract behavior [10, 11]. Moreover, most graph-based detectors were evaluated on a few bug classes, raising concerns about generalization to diverse vulnerabilities [10].

2.3 LLM-Driven Auditing and Hybrid Approaches

With the rise of large language models (LLMs), researchers have started applying them to smart contract auditing. Noever [19] evaluated GPT-4 against commercial analyzers and found it detected about four times more vulnerabilities while

suggesting valid fixes. However, Sovrano et al. [20] showed that LLMs exhibit position-based bias, missing vulnerabilities near the end of long code files. This highlights that naive LLM usage can be unreliable.

Several hybrid frameworks combine LLMs with structured reasoning or supporting modules. SmartGuard [21] augments LLMs with chain-of-thought prompting and code retrieval, achieving near-perfect recall but sometimes producing extraneous results. LLM-SmartAudit [22] uses a multi-agent strategy where specialized LLM agents collaborate, outperforming static analyzers on complex logical bugs. SmartLLMSentry [23] and Smart-LLaMA [24] fine-tune LLMs with domain-specific data and explanation-guided methods, improving interpretability and accuracy. Other systems, such as iAudit [25], mimic auditor workflows with multi-stage LLM pipelines.

Despite strong results, these methods still face challenges: reliance on large labeled datasets, prompt engineering, difficulty handling multiple subtle bugs, and false positives due to hallucination [22, 21]. Recent work therefore combines static analysis with LLM reasoning [26], which helps focus attention and validate findings.

2.4 Positioning of Our Work

Our work follows this frontier by leveraging LLMs within a modular auditing pipeline, integrating multiple scanning engines and an agentic validation mechanism. Unlike prior LLM-based auditing frameworks, which were mainly tested on benchmarks or simulated datasets, AuditCopilot is evaluated in real ongoing Web3 security audits. This real-world validation distinguishes our work, as only a few studies report live trials, such as limited internal audit pilots [16] or governmental payroll verification [17].

3 Design and Implementation

We present *AuditCopilot*, an LLM-based automated Web3 auditing system that integrates multiple scanning engines with an agentic validation framework. The system follows a modular pipeline architecture comprising five core components: project loader, planner, scanner, validator, and report generator. It automates Web3 project auditing through a sequential workflow where each component enriches the data for downstream processing:

The workflow mimics the auditing procedures in a real work setting. The core responsibility of each component is summarized below:

- *Loader*. Loads and parses the codebase of an audit project to a typed representation, which contains `Project`, `Commit`, `File`, and `Function` hierarchically.
- *Planner*. Serves as the core preprocessor to identify key files and functions and extract relevant context such as call graph and involved state variables. It can also select applicable checklist items for downstream scanning.

– *Scanner*. Performs function-level scanning and reasoning using structured prompts along with the context information extracted from the upstream planner.
– *Validator*. Conducts verifications using a multi-agent framework to validate preliminary findings and reduce false positives.
– *Reporter*. Exports findings in various formats (e.g., Markdown/Google Sheet) for auditor review.

The content below details the design and implementation of each of these components.

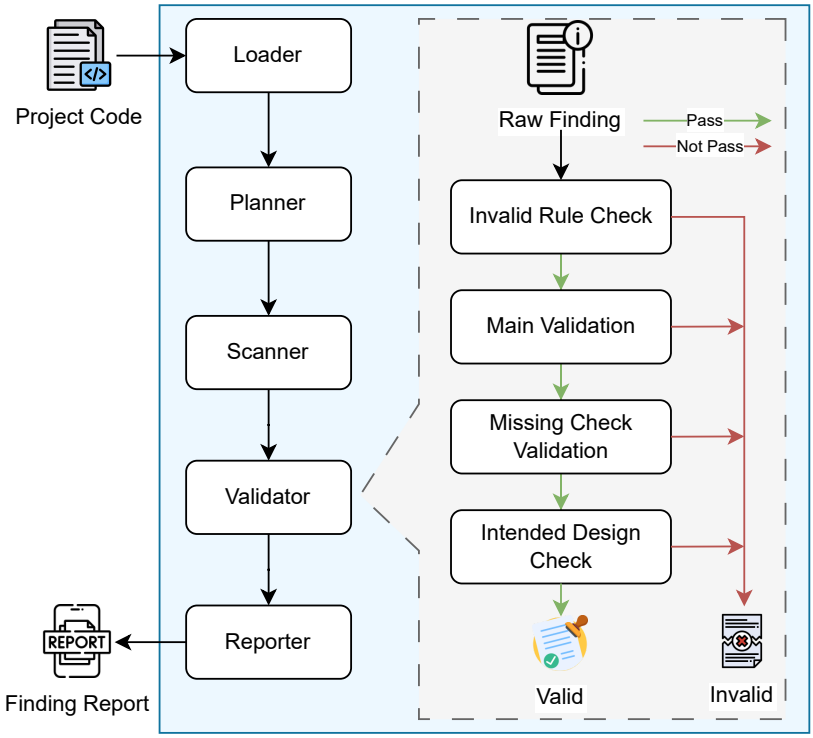


Fig. 1: An overview of AuditCopilot architecture.

3.1 Loader

The loader establishes the foundation by parsing project metadata and source code from either remote repositories or local file systems. It can set audit scopes

based on file extensions and manual scope lists, and create a hierarchical data model: `Project` → `Commit` → `File` → `Function`. Specifically, files and commits are loaded directly from the target project codebase. For functions, the loader uses language-agnostic, LLM-based parsing to extract only security-relevant (key) functions, while controlling analysis cost. The parsing stage is modular and can be replaced by static analysis tools (e.g., ANTLR4 [1]) to parse and extract functions and metadata when desired.

3.2 Planner

The planner plays an important role in extracting relevant information for downstream scanning and validation. It implements four specialized sub-modules:

- *File Locator*: Identifies key files, which usually contain core project logic, using LLM-based analysis of file structure and naming patterns.
- *Function Locator*: Extracts key functions in the identified key files with a focus on business logic, access control, state modifications, and external interactions.
- *Context Extractor*: Builds call graphs with configurable depth to capture context information of a given function. It also extracts relevant states and variables involved in the function.
- *Checklist Selector*: Chooses relevant security checklist categories based on function logic and behavioral patterns.

Each of these sub-modules uses LLM to perform its designed tasks. Specifically, the file locator and function locator leverage LLMs to identify and extract key functions from core logic files, as instructed in each of their prompts. To avoid overwhelming the scanner with less-vulnerable functions, blacklists and whitelists are used to filter certain files and functions. For example, the file locator excludes standard OpenZeppelin files (such as `AccessControl.sol` and `ReentrancyGuard.sol`) and the function locator excludes built-in functions (such as `revert` and `keccak256`) and utility functions that are not security-critical. This narrows down the scanning pool and saves scanning cost. The context extractor builds call graphs with configurable depth to capture function dependencies and data flow relationships using LLMs too. While it costs more than traditional methods, it provides broad support for projects in different languages. Note that a static parser generator can be used to replace the current LLM-based context extractor if cost sensitive. It also extracts relevant states and variables associated with each function, providing essential context for the scanner.

The checklist selector acts as a bridge to the checklist-driven scanning mode in the scanner. It examines the given function using LLM and selects relevant security checklist categories and items based on its logic and behavior. While LLMs can easily analyze and scan any given function, their outputs are often not deterministic and stable. The checklist addresses this by providing stable coverage of recurring vulnerability classes and aligning analysis with auditor

expectations. Therefore, we developed a curated security checklist that grounds LLM reasoning in a structured reference framework. In more detail, we collected over 12,000 findings from public audit reports since 2024 and used OpenAI's o1 and o3-mini models to abstract each finding into a candidate checklist item at a reusable, category-level granularity. These items were then grouped and deduplicated using automated clustering and AI screening, followed by our manual filtering and consolidation. Finally, we compiled 47 checklist categories including more than 1,500 distinct checks. Although it is not intended to be exhaustive, this list captures the most frequently observed and high-value security issues in practice, hence offering a precision-oriented reference that balances breadth with practicality. Table 1 shows some example checklist items from three categories.

Table 1: Example checklist items from the Input Validation, State Updates, and Ownership Transfer categories.

Category	**Checklist Item**
Input Validation	Ensure that functions processing multiple arrays enforce one-to-one correspondence by checking that the lengths of arrays (e.g., addresses and amounts) match. Verify that string inputs representing external addresses (e.g., BTC addresses) are properly validated or have documented off-chain validation to prevent malformed entries. Ensure that the `signature` parameter is exactly 65 bytes long before processing to prevent malformed signatures.
State Updates	Ensure that state updates occur atomically and in the correct sequence (e.g., update state before external calls) to follow the checks-effects-interactions pattern. Verify that all related state variables and counters are updated under all conditions (refund, deposit, withdrawal, rebalancing) to prevent locked or stale funds. Verify that refund, reversal, and cancellation functions consistently reset balances, counters, and flags to avoid stale state carry-over.
Ownership Transfer	Ensure ownership transfers use a two-step (pull-over-push) process where the new owner is nominated and must explicitly accept ownership before the transfer is finalized. Verify that ownership renunciation functions, like renounceOwnership, are either disabled or include strict safeguards (such as checking that the contract is not paused and resetting all related state variables) to prevent unintended loss of critical administrative control.

3.3 Scanner

The scanner component implements a multi-LLM scanning engine with adaptive scanning strategies based on project characteristics and task configurations. It supports regular and checklist-based scans. A regular scan is guided by a prompt and allows LLMs to explore vulnerabilities in depth, while a checklist-based scan leverages predefined security items to ensure coverage of common issue classes. Each mode can scan either the whole project or a single function as requested by the auditor. There are four scanning modes:

- *Regular Full Project Scan*: Analyzes all key functions with multiple models to maximize coverage.
- *Regular Single Function Scan*: Focuses on a single function as requested by the auditor, using multiple models for in-depth analysis.
- *Checklist-based Full Project Scan*: Leverages security checklists to guide the scanning process, ensuring all relevant areas are covered.
- *Checklist-based Single Function Scan*: Focuses on a single function as requested by the auditor and scans it with relevant checklist items.

Each of these scanning modes is backed by a structured prompt that guides the LLM to analyze the target function. For example, Listing 1.1 shows the core prompt used in the regular modes. The input includes the target function code, full source code of the file where the function is located, and its context extracted by the planner. The prompt is tuned to emphasize critical logical issues that usually cause fund losses for a project or its users. Our preliminary experiments show that LLMs tend to report many low-risk issues (such as coding style violations and gas optimizations) if not guided, which overwhelms auditors and wastes validation resources.

```
You are an expert security auditor with a focus on functional
programming in the Web3 space.

## Objective
Identify and clearly explain a specific, real logical vulnerability
within the provided Web3 function. The flaw must stem from a logic
error, not simply bad practices or technical deficiencies, and it
must be definite and exploitable. Target your explanation at human
auditors, ensuring accuracy and clarity.

## Instructions
1. **Vulnerability Identification**
   - Accurately describe the detected exploitable logical flaw
relevant to the supplied function-avoid generalizations,
recommendations, or unrelated issues.
2. **Exploit Demonstration**
   - Present a clear, practical, sequential example showing exactly
how the bug can be exploited using only the provided function.
Ensure the scenario is actionable, reproducible, and specific.
3. **Clarity and Precision**
```

```
   - Use concise and unambiguous explanations.
4. **Practicality**
   - Only analyze realistic, exploitable logic flaws. Ignore
scenarios requiring social engineering or privileged roles (such as
owner).
   - If reviewing ‘initialize‘, ignore ’first-caller wins’ as a
possible vulnerability.
   - Only consider reentrancy if it is clearly shown, not just a
hypothetical.
5. **Scope**
   - Limit your analysis to the target function and any provided
called functions. Exclude any external or unstated code.
6. **Roles**
   - Assume the attacker has no privileged roles. Owners and
privileged parties are honest; do not attribute mistakes to them.

After your analysis, briefly confirm that your identified issue fits
the criteria of a practical, actionable logical flaw within these
constraints.
If not, state that no valid vulnerability was found.
Always follow these instructions. Do not suggest fixes. Omit
analysis of code outside the defined scope. Ensure your analysis is
precise, actionable, and thorough.

## Output Format
Your output must be a valid AIRawFinding class instance with:
- ‘title‘: Descriptive, precise, and non-generic.
- ‘description‘: Clear summary of the vulnerability and concrete
exploitation steps.

## Provided Inputs
- **Input 1:** The target function. <function>{}</function>
- **Input 2:** List of functions called by the target.
<context>{}</context>
- **Input 3:** On-chain state/context.
<state_variable>{}</state_variable>
- **Input 4:** Full source code file for the target function. <code>
{} </code>
```

Listing 1.1: The core prompt used by the scanner in AuditCopilot.

The scanner adapts its scanning depth according to the size of the target project. For small projects, it carries out intensive analysis, typically scanning around forty functions with three independent passes for each function. This level of repetition allows the scanner to capture subtle issues and reduce the chance of overlooking rare vulnerabilities. In contrast, for large projects the focus shifts toward broader coverage rather than repeated deep scans. In such cases, the scanner usually examines around two hundred functions with a single pass each, ensuring that the analysis remains computationally efficient while still

providing meaningful coverage. This adaptive strategy strikes a balance between thoroughness and resource cost, making it suitable for both compact and large-scale codebases.

To further strengthen detection, the scanner relies on a multi-model setup that runs several large language models in parallel, such as GPT-5, Gemini Pro 2.5, and Grok 4. Each model evaluates the same set of functions independently, bringing in its own reasoning style and strengths. The outputs from these models are then combined, deduplicated, and validated later; this approach improves the system's ability to capture vulnerabilities that a single model might miss. By aggregating results across different reasoning engines, the scanner improves coverage and captures more potential vulnerabilities.

3.4 Validator

While the scanner is capable of generating many security findings, these results often include false positives. Unlike human auditors who can completely review code context, LLMs may have incomplete context understanding, incorrect interpretation of code semantics and core business logic, and hallucination effects. Consequently, these false positives can overwhelm auditors and reduce trust in automated analysis.

The goal of the validator is to act as a precision filter that screens raw scanner findings and identifies reliable security findings. Specifically, AuditCopilot is not designed to maximize the number of detected issues; instead, it focuses on suppressing false positives and ensuring that only high-confidence findings are included in the final report for auditor review. In this way, the validator shifts the framework from a detection-first approach to a validation-first approach. This design reflects the guiding philosophy of *precision over recall*. While traditional scanners often prioritize broad coverage to catch as many issues as possible, AuditCopilot takes a precision-focused stance. The objective is not to replace human auditors, but to assist them by delivering fewer yet more trustworthy findings. By minimizing false positives, the validator enables auditors to spend their time on real security issues rather than triaging false alarms.

The validator introduces a novel multi-stage approach to LLM-based finding validation. It is implemented as a directed workflow using LangGraph [2], where each stage acts as an autonomous agent with a specific validation role. This structured workflow enables the validator to combine complementary checks rather than relying on a single reasoning step. It reduces the likelihood of false positives and increases the reliability of accepted findings. By passing each candidate issue through multiple independent verification stages, the validator can separate true vulnerabilities from noise effectively. The complete validation workflow is illustrated on the right side in Fig. 1.

Validation Workflow The validation workflow consists of four specialized stages, each designed to address a different dimension of correctness and credibility. As shown on the right side in Fig. 1, this first validation stage, **Invalid**

Rules Check, removes false positive findings by applying a curated set of ten invalid finding rules. These rules capture common patterns where a reported issue does not represent a real vulnerability, such as assuming attacker control over privileged roles, relying on unrealistic owner mistakes, or describing impractical overflow/underflow cases. They also cover benign conditions like fee-on-transfer behavior, "first-caller wins" initialization, or findings that require external manipulation of users. Practically, these rules were created and accumulated by auditors from their experience triaging false positives in real audits. By codifying these invalid scenarios, it quickly suppresses noise such as unrealistic assumptions, unpractical exploit scenarios, and stylistic warnings, which are frequently considered as false positives in real audit settings.

After filtering out findings based on these rules, each remaining finding goes through the **Main Validation** stage. This stage uses a carefully designed prompt to instruct LLMs to mirror how a human auditor would usually review findings. Specifically, the LLM model is instructed to critically verify whether a reported issue is real, practical, and exploitable in context, rather than theoretical or vague. This validation step enforces a high bar for acceptance, as the goal is to achieve a high precision score. The model must examine the finding in relation to the function, its called dependencies, on-chain state, and even broader project structure. It only marks a finding as valid if the evidence is clear, precise, and realistic from an attacker's perspective. By emphasizing practicality and requiring strong confidence in each verdict, this step significantly reduces false positives while preserving true issues for human auditors to review. To support this process, the validator is provided with tools that mimic how a human auditor cross-checks information. For example, the local file search tool enables direct search of the project codebase; the vector store search tool provides semantic retrieval across large repositories, and the web search offers online search capability such as protocol specifications or known vulnerability references. These tools allow the model to gather missing details, verify assumptions, and form a complete understanding of the finding, making the validation both thorough and reliable.

Next, the **Missing Check Validation** stage specifically targets two categories of findings that LLMs often report but turn out to be false positives: access control and input validation. In our preliminary experiments, we found that LLMs frequently misinterpret whether such checks are present or absent. For example, a model may report an input validation issue in a function while overlooking that the input is already sanitized in a higher-level caller (often in a function several levels up). Similarly, it may incorrectly flag missing access control because it fails to trace the enforcement across multiple layers of the codebase. To address these cases, this stage explicitly analyzes the propagation of access control and input validation logic across the full execution flow. By reasoning about how checks are applied within and across functions, it prevents false positives that arise from incomplete or shallow inspection. As in the main validation stage, the same three types of tools are enabled to the model for gath-

ering relevant context and references across the entire project, just as a human auditor would.

Lastly, the **Intended Design Check** addresses a common source of false positives: findings that are not vulnerabilities but rather part of the project's intended design. In practice, many reported issues, e.g., unrestricted transfer functions in ERC20/ERC721 standards or game-specific mechanics, may appear risky in isolation but are deliberate features of the system. To handle such cases, this stage prompts the model to evaluate whether the flagged behavior aligns with established standards, project documentation, or common design practices. Similar to how a human auditor works, the model consults multiple sources of evidence, including local project files, uploaded references (e.g., whitepapers, design documents), and online resources such as official GitHub repositories or protocol specifications. By cross-referencing findings against authoritative references, it can distinguish genuine vulnerabilities from intentional features, ensuring that only true issues are escalated to human auditors.

To summarize, the workflow mirrors a human auditor: gather context, move back and forth between code and references, and use tools to confirm doubts. It cycles through read → verify → cross-reference with three tools to keep precision high. It is worth noting that we also built an alternative validator that treats LLM models as independent judges. Each judge reviews the finding using the same prompt and returns a verdict and brief reason. We run multiple models and multiple rounds per model, then aggregate results by majority with a high threshold (e.g., $>= 0.8$ agreement, so at least 5/6 or 7/8 judges). This produces a single, aggregated verdict and a representative reason. However, this approach behaves like one-shot reasoning: it does not iteratively build context or query tools during review. In practice, its precision lags behind our staged, tool-driven validator. We include a head-to-head comparison as part of our evaluation in Section 4.2.

3.5 Reporter

The reporter component is used to generate the final finding report based on the validation results. It is designed to provide easy-to-read and actionable findings for auditors to review. For example, it supports multiple output formats, including Markdown reports and Google Sheets, to accommodate different auditor preferences. Each finding in the report includes detailed metadata such as the function name, file location, commit hash, scanner model, validation trace, and severity level.

3.6 Implementation Details

AuditCopilot is implemented as a modular Python framework built on Flask, with each core component exposed through RESTful APIs for scalable deployment and integration with external audit platforms. The system implements a hierarchical data model following the structure `Project` → `Commit` → `File` → `Function`, The framework employs parallel processing for key operations and

leverages the LangChain and LangGraph agents framework to orchestrate multi-step reasoning workflows in both scanner and validator components. The agentic validator operates as a directed workflow where each validation stage acts as an autonomous agent with specialized tools for context gathering, of which the goal is to mirror human auditor practices. The system integrates multiple LLM providers, including OpenAI (GPT-5), Google Gemini (Gemini Pro 2.5), and xAI (Grok 4), to enhance its reasoning capabilities and provide diverse perspectives on the code being audited.

4 Evaluation

We evaluate AuditCopilot in collaboration with a professional Web3 auditing company[3], using their real ongoing auditing projects as the testbed. This setup allows us to measure the system not only in a controlled environment but also under practical conditions where findings must be triaged by experienced auditors. Every output generated by AuditCopilot was reviewed by auditors, who determined whether a reported issue was valid. This evaluation process reflects how AuditCopilot would be adopted in practice, and ensures that the reported performance metrics are meaningful to real-world auditing workflows. Specifically, we answer the following research questions to evaluate the effectiveness of AuditCopilot in the following subsections:

- **RQ1**: How effective is AuditCopilot in real-world audits compared to baseline LLM approaches?
- **RQ2**: Does the agentic validator improve precision compared to baseline methods?
- **RQ3**: How does each stage of the agentic validator contribute to reducing false positives?

4.1 RQ1: Effectiveness in Real-World Audits

To answer RQ1, we evaluate AuditCopilot on 14 real audit projects provided by the Web3 auditing company from August 1 to August 31, 2025. Each project was scanned independently using OpenAI GPT-5 and Gemini Pro 2.5, and validated using OpenAI GPT-5. After validation, all reported findings were triaged by professional auditors to determine their validity.

As shown in Table 2, the 14 scanned projects vary widely in size and complexity, ranging from tiny contracts with only one or two files to large-scale codebases containing hundreds or even thousands of files. This diversity is important for evaluation, as it mirrors the real conditions faced by auditors and

[3] The company asked not to be named due to confidentiality agreements with its clients and several ongoing audits during the study period. It is one of the top Web3 security global firms with experienced full-time auditors. Only aggregate, de-identified results are reported, and anonymity does not change the validation process: every finding was still manually reviewed by senior auditors.

tests whether the system can adapt its performance across very different project profiles. On average, the cost of scanning each project was around 21 USD, which is affordable compared to the thousands USD manual auditing fees.

Table 2: Per-project scan and evaluation details.

Project Details			Scanner Details			Validator Details	
Name	**Total Files**	**In-Scope Files**	**Key Files**	**Key Functions**	**Total Findings**	**Reported Findings**	**Precision**
A	2	2	2	12	24	5	80.0%
B	105	25	25	54	108	9	88.9%
C	49	5	5	17	34	2	50.0%
D	210	25	25	56	112	9	88.9%
E	6	6	6	23	46	5	40.0%
F	32	7	7	17	34	3	100.0%
G	1	1	1	9	18	1	0.0%
H	1	1	1	5	10	3	66.7%
I	4	4	4	16	32	1	100.0%
J	2,511	11	11	21	42	2	0.0%
K	1	1	1	10	20	2	100.0%
L	60	7	7	16	32	1	0.0%
M	9	3	3	7	14	1	100.0%
N	1	1	1	10	20	1	100.0%
Total	3,992	123	98	268	642	45	73.3%

On the scanner side, the pipeline first narrows down the search space. From the full codebase, clients often define a set of *in-scope* files, which are the files explicitly requested for auditing. These are often much smaller than the entire project, yet context from the whole codebase is still considered to ensure dependencies and external calls are properly analyzed. From the in-scope set, the planner component then selects *key files*, and within them identifies *key functions* that concentrate most of the potential security logic. For example, *project B* had 25 in-scope files but 54 key functions selected, while *project E* had 6 in-scope files and 23 key functions. Even very small projects followed this process, such as *project A* with only 2 in-scope files and 12 key functions. Each key function was then scanned independently by two complementary LLMs: OpenAI GPT-5 and Gemini Pro 2.5. This dual-model strategy increased coverage and reduced blind spots, since the models have different reasoning and detection strengths. The results of these scans formed the pool of *total findings*, which the validator later filtered down to high-confidence reports for auditors.

On the validator side, the effect of filtering is clear from both Figure 2 and Table 2. While the scanner produced a large pool of raw candidates (642 total

raw findings across all 14 projects), only a small fraction passed the validator pipeline. In more details, AuditCopilot finally reported 45 findings which were later triaged by different auditors. Out of these, 33 were confirmed as valid security issues while 12 were rejected as false positives, yielding an overall precision of 73.3%. This also indicates that less than 7% of all raw findings were forwarded for human review, showing the validator's strong noise suppression capability. The median number of reported findings per project was 2. This indicates that in practice auditors reviewed only a handful of items per audit project. This matches our goal of producing "fewer, higher-confidence" findings.

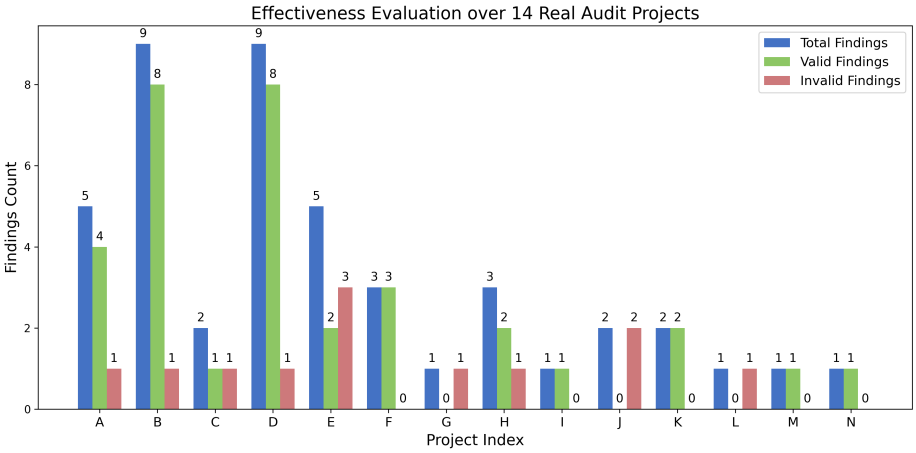
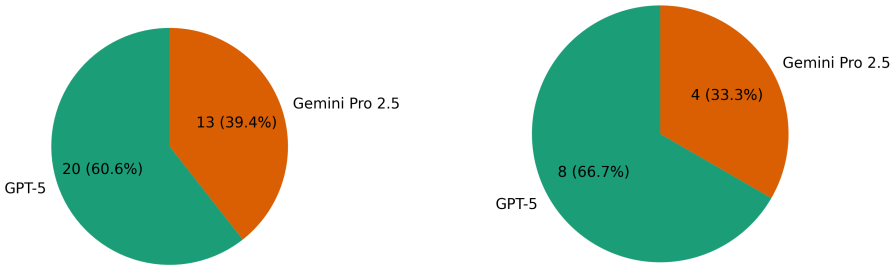


Fig. 2: Overall performance of AuditCopilot across 14 real audit projects.

The per-project results highlight and confirm the effectiveness of AuditCopilot. For example, in *project D*, the scanner produced 112 raw findings, but the validator reduced these to only 9 reported findings, of which 8 were valid (88.9%). Similarly, *project A* started with 24 scanner findings but reported only 5, and 4 of them were valid (80.0%). Even in small projects, the validator played a crucial role: in *project N*, 20 raw findings were screened to just one reported issue, which was fully valid. Although a few cases like *project L* yielded no validated findings, the overall trend demonstrates that the system reliably contributes meaningful vulnerabilities across diverse engagements. Taken together, the numbers show that the validator acts as the precision engine of the workflow. It consistently reduces a noisy candidate set of hundreds of findings down to a handful of high-confidence, auditor-ready reports.

As shown in Fig. 3, both OpenAI GPT-5 and Gemini Pro 2.5 contributed meaningfully to the final set of reported issues. Across all projects, OpenAI GPT-5 reported 27 findings, of which 20 were validated as true issues (74.1% precision). Gemini Pro 2.5 reported 16 findings, with 13 validated (81.3% precision). Together, the two models accounted for all 33 valid findings. This confirms

that running them in parallel is beneficial: OpenAI GPT-5 provided broader coverage, while Gemini offered slightly higher precision. The combination reduced blind spots and balanced recall with reliability.



(a) Distribution of Valid Findings (b) Distribution of Invalid Findings

Fig. 3: Breakdown of GPT-5 and Gemini Pro 2.5: (a) valid findings, (b) invalid findings.

```
Title: Pre-start cancellation makes toTime < lastRewardUpdatedAt,
causing _updatePool underflow and locking staked tokens

Description:
  The _updatePool function assumes the effective end time (toTime)
is always >= lastRewardUpdatedAt. This is false if a pool is
cancelled before the scheduled reward start. When rewards were
scheduled to start in the future (pre-staking case), rewardStartedAt
and lastRewardUpdatedAt are set to rewardStartsAt (future). If the
creator cancels the pool before that time, cancelledAt <
rewardStartedAt. Later, once block.timestamp surpasses
rewardStartedAt and any function calls _updatePool, the function
computes:
  - endTime = rewardStartedAt + rewardDuration,
  - then endTime = cancelledAt (because cancelledAt < endTime)
  - toTime = min(currentTime, endTime) = cancelledAt
  - timePassed = toTime - lastRewardUpdatedAt = cancelledAt -
rewardStartedAt
  This subtraction underflows in Solidity 0.8+, reverting every call
that reaches _updatePool. Because unstake, emergencyUnstake, and
claim all call _updatePool first, users are blocked from unstaking
or claiming; their staking tokens are locked in the contract.

Exploit demonstration:
```

```
  (1) Create a pool with rewardStartsAt = now + X (X up to 7 days is
allowed) and a valid rewardDuration.
  (2) A user pre-stakes into the pool (first stake). This sets
rewardStartedAt = rewardStartsAt and lastRewardUpdatedAt =
rewardStartsAt.
  (3) Before rewardStartsAt is reached, call cancelPool for that
pool. Internally, _updatePool returns early (currentTime <=
lastRewardUpdatedAt). cancelPool sets cancelledAt = currentTime (<
rewardStartedAt) and returns all leftover rewards to the creator.
  (4) After rewardStartsAt passes, any call that triggers
_updatePool (claim, unstake, or emergencyUnstake) will revert at:
uint256 timePassed = toTime - lastRewardUpdatedAt; because toTime =
cancelledAt < lastRewardUpdatedAt.
  (5) Result: All stakers who did not unstake before rewardStartsAt
are permanently unable to unstake or claim; their staked tokens are
locked in the contract. This is a direct consequence of the logical
flaw in _updatePool's time window computation when cancelledAt <
rewardStartedAt.
```

Listing 1.2: An example critical finding found by AuditCopilot.

Listing 1.2 shows one example of a critical finding reported by AuditCopilot and later confirmed by auditors. It is a time ordering logic flaw where a creator cancels a staking pool before rewards start; this makes the later reward update routine assume an impossible time window. This case required reasoning across an initialization phase, a premature cancellation path, and a delayed settlement call, rather than a single local pattern. The scanner produced the candidate by tracing the state variables that change before and after cancellation; the validator kept it because the exploit path was concrete, did not rely on privileged misuse, and had a clear, high-impact effect. Auditors classified it as critical severity since it can trap user assets with no on-chain recovery path.

4.2 RQ2: Precision Improvement with the Agentic Validator

To address RQ2, we compare the agentic validator against two baseline approaches: a vanilla LLM and the earlier LLM judge. The vanilla LLM setup is the simplest: we provide LLM with a prompt that includes the function and the reported finding, and ask it directly whether the finding is valid. The LLM judge improves on this by involving three different models, each asked to validate the same finding twice. The input prompt is richer since it contains the function, the reported finding, and additional context extracted by the planner. The results are then aggregated, and a finding is marked valid only if all judges agree. However, this process still amounts to a single-round judgment. In contrast, the agentic validator follows a multi-step workflow more similar to a human auditor. It gathers sufficient context, checks supporting evidence, and uses external tools when necessary before making a decision. This structured approach is designed to suppress false positives and improve precision.

For this evaluation, we selected a medium-sized real audit project containing 108 Solidity files. From this project, the scanner identified 25 key files and 91 key functions. Using two different models within the scanner as described in Section 3.3, it generated 181 raw findings. We then applied three validation strategies:

- **Vanilla LLM**: using OpenAI's GPT-5 with a simple prompt containing the function and the reported finding.
- **LLM Judges**: using three models, i.e., OpenAI GPT-5, OpenAI o3, Gemini Pro 2.5, each run twice, with the prompt including function, finding, and planner-extracted context. A strict threshold of 1 was used, meaning all judges had to agree for a finding to be marked valid.
- **Agentic Validator**: applying the multi-step workflow as described in Section 3.4.

Note that the LLM judge was our earlier version of the validator implemented in AuditCopilot. We tested it in several real audits and observed that, although it improved precision compared to vanilla LLMs, it still produced too many false positives. This limitation motivated the design of the current agentic validator. We include the LLM judge in our evaluation to illustrate the evolution of our validation strategies and how each step leads to higher precision. Table 3 summarizes the results.

Table 3: Comparison of validation strategies on a real audit project (108 files, 181 raw findings).

Method	**Raw Findings**	**Reported Findings**	**Valid (Triaged)**	**Invalid (Triaged)**	**Precision**
Vanilla LLM (GPT-5)	181	61	15	46	24.6%
LLM Judges	181	20	11	9	55.0%
Agentic Validator	181	8	7	1	87.5%

These results show a clear progression in precision. After validating those 181 raw scanner findings, the vanilla LLM reported 61 findings, but only 15 were valid based on auditor triage, leading to a low precision of 24.6%. The LLM judge cut the reports to 20 findings, with 11 valid and 9 invalid, raising precision to 55.0% but still retaining many false positives. The agentic validator reported only 8 findings, of which 7 were valid, reaching 87.5% precision. While this means it missed 8 valid cases compared to the vanilla LLM and 4 compared to the LLM judge, this outcome is by design: the agentic validator filters aggressively to suppress false positives and only keeps findings with strong supporting evidence. In practice, this precision-oriented strategy reduces the manual review burden for auditors, and any missed issues could be complemented by traditional static analyzers such as Slither or by internal company tools, ensuring that recall is preserved at the workflow level while precision is maximized.

4.3 RQ3: Contribution of Each Validator Stage

To answer RQ3, we analyze how each stage of the agentic validator contributes to reducing false positives in the aforementioned 14 real audit projects. Figure 4 shows how the agentic validator processes the 642 raw scanner findings through its four stages. The first stage, the invalid rules check, eliminates 180 findings (about 28%), showing that a set of ten rule-based filters can already prevent a large share of obvious false alarms from reaching the more expensive validation stages. The main validation stage then has the strongest impact: of the 462 findings that passed the initial validation stage, 265 (57%) were rejected after detailed reasoning. This step alone accounts for nearly half of all invalidations, confirming that LLM-guided reasoning backed with various tools is the core driver of precision gains. The later stages play a smaller but still important role.

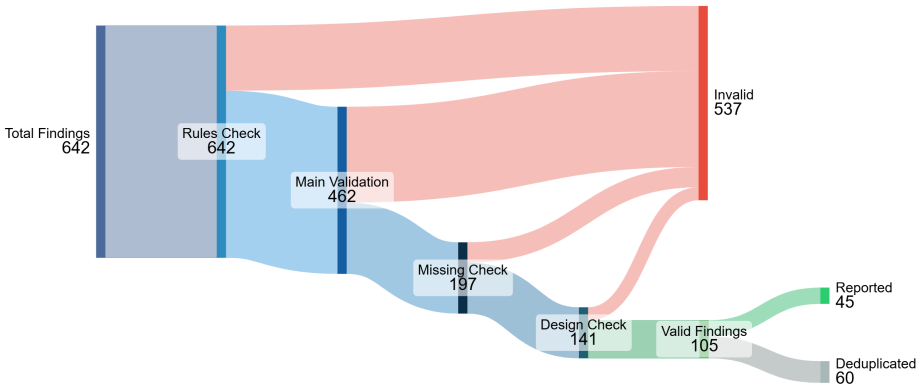


Fig. 4: Effectiveness of each stage in the agentic validator.

The missing-check validation removed 56 findings caused by incomplete or partial checks (with emphasis on access control and input validation across callers and callees), while the intended-design validation suppressed 36 findings that reflected expected behavior rather than real vulnerabilities. Although these numbers are smaller in absolute terms, they target nuanced cases that neither rules nor test-based reasoning could catch, further refining the quality of the output report for auditors. After all filters, 105 findings remained valid, corresponding to only 16% of the original pool. Once duplicates were merged at the file and function levels, it eventually reported 45 unique findings, as presented in Section 4.1.

From another perspective, if the validator had stopped earlier, precision would have been substantially lower. For example, skipping the design and missing-check validations would reduce precision from 74% to 53%, while re-

moving the main validation would drop it below 23%. Without any filtering, precision would have been only 16%. These comparisons highlight that every stage contributes meaningfully to overall precision, with the main validation responsible for the largest single gain and the later checks ensuring that subtle false positives do not persist.

In summary, these results show that the validator's multi-stage design is effective not only at discarding clear false alarms early, but also at refining borderline cases through deeper reasoning and design-aware checks. The final deduplication step then condenses repeated signals into concise, actionable reports. Overall, the pipeline reduces 642 initial findings to just 45 unique validated issues, demonstrating how layered validation can achieve both high precision and practical report quality.

5 Discussion

Our design of AuditCopilot emphasized precision over recall, and this trade-off defines both the strengths and limits of the current system. The validator is intentionally strict, as shown in RQ2 (Section 4.2), where the agentic validator reported the fewest findings but achieved the highest precision. In practice, this means that while auditors receive fewer false alarms, the system may miss valid issues that less selective baselines are able to capture. Across the 14 projects we tested, AuditCopilot reached 73.3% precision, but we did not evaluate recall against a full ground truth, leaving open the question of how many real vulnerabilities were suppressed. This reflects the central limitation: high confidence in reported findings comes at the cost of incomplete coverage. The system also inherits challenges from its use of multiple LLMs. Outputs can drift when models are updated, which may affect reproducibility of results across time. In addition, the cost and latency of multi-model scanning and validation grow with project size, making scalability an open concern.

Looking forward, several directions could strengthen both coverage and usability. One important step is to complement the precision-oriented validator with category-specific scanners. Instead of relying on a single general-purpose scan, dedicated scanners could target each vulnerability category in the checklist, using prompts and tests tailored to that class. This would expand recall and coverage without discarding the validator's strict filtering. A hybrid approach that combines LLM reasoning with static analysis, symbolic execution, and fuzzing could also help address gaps in dynamic validation. By routing findings to the most appropriate tool, the system can improve its ability to confirm issues that are difficult to reproduce with LLM-generated tests alone.

In summary, while AuditCopilot demonstrates that precision-focused LLM pipelines can meaningfully reduce false positives in Web3 audits, its limits in recall, coverage, and scalability leave many opportunities for future improvement. By combining stricter validation with broader scanning, deeper testing, and ecosystem-specific extensions, future work can balance precision with re-

call and make LLM-driven auditing a more comprehensive tool for the security community.

6 Conclusion

In this paper, we present AuditCopilot, which shows that a precision-first, LLM-driven workflow can make Web3 audits faster and more reliable. Our modular pipeline, two scan modes, and 47-category checklist feed a four-stage validator that cuts false alarms before they reach auditors. Across 14 real audit projects, the scanner generated 642 raw finding candidates; the validator reported 45 of them, of which auditors confirmed 33 as valid, yielding 73.3% precision. In a head-to-head on one medium project, the agentic validator reached 87.5% precision (7 of 8 reported were valid), outperforming a vanilla LLM and an LLM-judges setup. These results match our design goal: fewer, clearer, and higher-confidence findings that reduce triage effort and support auditors for real-world audits.

References

1. ANTLR4: ANother Tool for Language Recognition. GitHub (2025). https://github.com/antlr/antlr4
2. LangGraph: Build resilient language agents as graphs. GitHub (2025). https://github.com/langchain-ai/langgraph
3. Luu, L., Chu, D.-H., Olickel, H., Saxena, P., Hobor, A.: Making Smart Contracts Smarter. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS) (2016)
4. Mythril: Security Analysis Tool for Ethereum Smart Contracts. GitHub (2018). https://github.com/ConsenSys/mythril
5. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., Vechev, M.: Securify: Practical Security Analysis of Smart Contracts. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS) (2018)
6. Feist, J., Grieco, G., Groce, A.: Slither: A Static Analysis Framework for Smart Contracts. In: IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB) (2019)
7. Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhteyev, A., Marchenko, E., Alexandrov, Y.: SmartCheck: Static Analysis of Ethereum Smart Contracts. In: 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (2018)
8. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: Analyzing Safety of Smart Contracts. In: Network and Distributed System Security Symposium (NDSS) (2018)
9. Qian, P., Zhang, Z., Yu, Y., Wu, L.: Towards Automated Reentrancy Detection in Smart Contracts. IEEE Access **8**, 19685–19695 (2020)
10. Nguyen, T.-D., Pham, H., Sun, J., Lin, Y., Xu, J.: MANDO: A Graph Neural Network-Based Smart Contract Vulnerability Detection System. In: Proceedings of the 44th International Conference on Software Engineering (ICSE) (2022)
11. Wu, H., Feng, C., Chen, Q., Luo, X.: Peculiar: Smart Contract Vulnerability Detection via Multi-View Graph Learning. In: Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI) (2021)

12. Jiang, Y., Liu, H., Wang, H.: VDDL: Dependency Graph Based Vulnerability Detection in Smart Contracts. Journal of Information Security and Applications **65**, 103081 (2022)
13. Deng, K., Xu, Z., Huang, J., Chen, Y.: A Multimodal Deep Learning Approach for Smart Contract Vulnerability Detection. IEEE Transactions on Dependable and Secure Computing (2023)
14. Yang, Z., Man, G., Yue, S.: Automated Smart Contract Vulnerability Detection using Fine-tuned Large Language Models. In: 2022 5th International Conference on Blockchain Technology and Applications (2023)
15. Yang, Z., Man, G., Yue, S.: Understanding Security Audits on Blockchain. In: 2022 5th International Conference on Blockchain Technology and Applications (2022)
16. Emett, S., Eulerich, M., Lipinski, E., Prien, N. and Wood, D.A., 2025. Leveraging ChatGPT for enhancing the internal audit process—a real-world example from Uniper, a large multinational company. Accounting Horizons (2025)
17. Li, X., Zhang, Y., Chen, H.: Continuous Audit Cross-Verification with LLMs: A Case Study on Government Payroll Systems. In: International Conference on E-Governance and Information Systems Security (2024)
18. Tang, X., Wang, Z., Li, M., Liu, H.: LightningCat: A CodeBERT-Based Approach for Smart Contract Vulnerability Detection. Applied Sciences **13**(3), 1455 (2023)
19. Noever, D.: GPT-4 vs. Static Analyzers: Detecting Smart Contract Vulnerabilities with Large Language Models. arXiv:2304.01245 (2023)
20. Sovrano, F., Martino, A., Russo, G.: Bias in LLM-Based Code Analysis: A Study on Position-Sensitivity. In: IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) (2024)
21. Ding, W., Zhang, H., Liu, J.: SmartGuard: Augmenting Large Language Models with Retrieval and Reasoning for Smart Contract Security. In: ACM Symposium on Blockchain (2025)
22. Wei, Y., Wang, F., Chen, X., Zhao, Y.: LLM-SmartAudit: Multi-Agent Collaboration for Smart Contract Auditing. In: International Conference on Autonomous Agents and Multiagent Systems (AAMAS) (2024)
23. Zaazaa, H., El Bakkali, H.: SmartLLMSentry: In-Context Learning for Smart Contract Vulnerability Detection. Future Internet **16**(2), 35 (2024)
24. Yu, B., Guo, L., Zhang, H.: Smart-LLaMA: Domain-Aware LLM for Smart Contract Security Auditing. Electronics **13**(6), 1234 (2024)
25. Ma, J., Zhou, C., Liu, K.: iAudit: Two-Stage LLM-Based Framework for Smart Contract Auditing. In: IEEE International Conference on Blockchain (2024)
26. Zhou, R., Yang, Z., Li, P.: GPTScan: Hybrid Static and LLM-Driven Analysis for Smart Contract Security. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS) (2024)