SmartAuditFlow: A Dynamic Plan-Execute Framework for Advanced Smart Contract Security Analysis

ZHIYUAN WEI, Beijing Institute of Technology, China
JING SUN, University of Auckland, New Zealand
ZHE HOU, Griffith University, Australia
ZIJIAN ZHANG*, Beijing Institute of Technology, China
ZIXIAO ZHAO, University of Auckland, New Zealand
CHUNMIAO LI, Beijing Academy of Blockchain and Edge Computing, China
MINGCHAO WAN, Beijing Academy of Blockchain and Edge Computing, China
JIN DONG, Beijing Academy of Blockchain and Edge Computing, China

Large Language Models (LLMs) have demonstrated significant potential in smart contract auditing. However, they are still susceptible to hallucinations and limited context-aware reasoning. In this paper, we propose SmartAuditFlow, a dynamic Plan-Execute framework that customizes audit strategies based on the unique characteristics of each smart contract. Unlike static, rule-based workflows, our approach iteratively generates and refines audit plans in response to intermediate outputs and newly detected vulnerabilities. To improve reliability, the framework incorporates structured reasoning, prompt optimization, and external tools such as static analyzers and Retrieval-Augmented Generation (RAG). This multi-layered design reduces false positives and enhances the accuracy of vulnerability detection. Experimental results show that SmartAuditFlow achieves 100% accuracy on common vulnerability benchmarks and successfully identifies 13 additional CVEs missed by existing methods. These findings underscore the framework's adaptability, precision, and practical utility as a robust solution for automated smart contract security auditing. The source code is available at: https://github.com/JimmyLin-afk/SmartAuditFlow.

CCS Concepts: • **Security and privacy → Software security engineering**; • **Computing methodologies → Planning and scheduling**.

Additional Key Words and Phrases: smart contract, LLM, code auditing, RAG, prompt optimization

ACM Reference Format:
Zhiyuan Wei, Jing Sun, Zhe Hou, Zijian Zhang, Zixiao Zhao, Chunmiao Li, Mingchao Wan, and Jin Dong. 2025. SmartAuditFlow: A Dynamic Plan-Execute Framework for Advanced Smart Contract Security Analysis. *ACM Trans. Softw. Eng. Methodol.* 37, 4, Article 111 (August 2025), 43 pages. https://doi.org/XXXXXXX.XXXXXXX

*Corresponding author.

Authors' Contact Information: Zhiyuan Wei, Beijing Institute of Technology, Beijing, Beijing, China, weizhiyuan@bit.edu.cn; Jing Sun, University of Auckland, Auckland, New Zealand, jing.sun@auckland.ac.nz; Zhe Hou, Griffith University, Queensland, Australia, Z.hou@griffith.edu.au; Zijian Zhang, Beijing Institute of Technology, Beijing, Beijing, China, zhangzijian@bit.edu.cn; Zixiao Zhao, University of Auckland, Auckland, New Zealand, zixiao.zhao@auckland.ac.nz; Chunmiao Li, Beijing Academy of Blockchain and Edge Computing, Beijing, China, chunmiaoli1993@gmail.com; Mingchao Wan, Beijing Academy of Blockchain and Edge Computing, Beijing, China, chainmaker@baec.org.cn; Jin Dong, Beijing Academy of Blockchain and Edge Computing, Beijing, China, dongjin@baec.org.cn.

Manuscript submitted to ACM

1 Introduction

Smart contracts, foundational to blockchain networks and decentralized finance (DeFi), have frequently exhibited critical security vulnerabilities, posing significant financial and operational risks. The inherent complexity of smart contract security is a challenge, particularly as developers may lack comprehensive means to assess security rigorously prior to deployment. Unlike traditional software, where post-deployment patching is standard, the immutable or costly-to-modify nature of deployed smart contracts means that vulnerabilities can lead to substantial and often irreversible losses across temporal, operational, financial, and reputational dimensions. Several high-profile incidents have underscored these vulnerabilities, such as the DAO Exploit (50 million USD), Poly Network Exploit (610 million USD), and Bybit Crypto Exchange Hack (1.5 billion USD). In 2024 alone, financial losses from smart contract vulnerabilities exceeded **$2.6 billion** USD across **192** incidents, resulting from smart contract-level vulnerabilities[1]. These incidents underscore the critical and urgent need for robust security auditing mechanisms.

Existing smart contract security auditing primarily relies on manual code reviews by security experts, often supplemented by automated analysis tools [43]. While valuable, these approaches face significant limitations in the fast-paced blockchain environment. Manual audits, despite their potential for depth, are inherently time-consuming and costly, often creating bottlenecks in development cycles. Furthermore, the continuous evolution of blockchain technology and the emergence of novel attack vectors challenge the ability of human auditors to consistently identify all vulnerabilities [30]. Expert reviews also rely on individual experience and knowledge, which can introduce subjectivity and gaps in coverage. Conversely, conventional automated tools, while offering speed, frequently suffer from high false positive rates and struggle to detect complex, context-dependent vulnerabilities. Such tools often lack the nuanced understanding of business logic and intricate blockchain-specific security patterns required to identify subtle flaws, such as sophisticated logic errors or vulnerabilities arising from inter-contract interactions, thereby limiting their overall effectiveness.

The advent of Large Language Models (LLMs) presents a promising avenue for advancing smart contract auditing and addressing the aforementioned limitations [25]. LLMs possess the unique ability to process and understand both natural language (e.g., contract specifications, comments) and programming languages (i.e., smart contract code) [22]. This facilitates a deeper reasoning about the alignment between a contract's intended functionality and its actual implementation. Their notable capabilities in pattern recognition and contextual analysis are particularly valuable for detecting subtle or novel security vulnerabilities that traditional automated tools might overlook. Furthermore, LLMs are good at reasoning processes and explaining identified security issues in an accessible manner, thereby empowering developers who may not be security specialists to understand and address the findings.

Recent research has shown encouraging results in applying LLMs to smart contract security analysis [9, 37]. Nevertheless, substantial challenges hinder the development of fully reliable and effective LLM-based auditing systems. Key issues include: (1) output inconsistency and non-determinism [3]; (2) the difficulty of validating the correctness and completeness of LLM-generated security assessments; and (3) limitations in ensuring comprehensive exploration of potential vulnerabilities. These factors necessitate careful strategies to harness LLM strengths while mitigating their inherent weaknesses. These challenges can be effectively addressed by adopting a **workflow-based** approach. This methodology decomposes the complex auditing task into manageable, discrete steps, facilitating structured and iterative interactions with LLMs. The efficacy of such structured methods is evident in the improved performance of advanced LLMs in complex reasoning when guided by systematic procedures [44], and is central to agentic workflows like HuggingGPT, Manus, and AutoGPT [34], which orchestrate LLMs for sophisticated problem-solving.

[1] https://getfailsafe.com/failsafe-web3-security-report-2025/

In this paper, we introduce SmartAuditFlow, a novel framework that integrates dynamic audit plan generation with workflow-driven execution for robust smart contract auditing. It systematically structures the auditing process by (1) in-depth analysis of the smart contract's structural and functional characteristics, (2) generation of an adaptive audit plan, (3) iterative execution of this plan via a multi-step workflow, and (4) progressive refinement of security assessments. By emulating the methodical approach and adaptability of expert human auditing teams, SmartAuditFlow aims to significantly enhance the accuracy, efficiency, and comprehensive security analysis. Our primary contributions are as follows:

- **Dynamic and Adaptive Audit Plan Generation:** We propose a novel framework for generating audit plans that are dynamically tailored to the specific characteristics (e.g., complexity, potential risk areas identified through static analysis) of each smart contract. This plan adaptively refines its audit strategies based on intermediate findings from LLM analysis, leading to a more targeted and exhaustive security assessment.
- **Iterative Prompt Optimization:** We introduce a dynamic prompt engineering strategy where prompts are iteratively refined throughout the audit lifecycle. This refinement is based on real-time analytical feedback and the evolving audit context, enhancing LLM accuracy, improving the reliability of security assessments, and mitigating common issues such as output misinterpretation or factual hallucination.
- **Integration of External Knowledge Sources:** SmartAuditFlow incorporates a hybrid approach by integrating inputs from traditional static analysis tools (for extracting key code attributes and potential hotspots) and Retrieval-Augmented Generation (RAG). The RAG component leverages external knowledge bases to provide LLMs with up-to-date, authoritative security information and vulnerability patterns, thereby enriching contextual understanding and improving the accuracy and completeness of detected vulnerabilities.
- **Rigorous Evaluation and Efficiency Improvements:** We conduct a comprehensive empirical evaluation of SmartAuditFlow across a diverse set of smart contracts using multiple performance metrics, including Top-N Accuracy, Mean Reciprocal Rank (MRR), and Mean Average Precision (MAP). Our results demonstrate that SmartAuditFlow outperforms traditional static analysis tools and existing LLM-based auditing frameworks, detecting a broader spectrum of vulnerabilities while significantly improving computational efficiency and scalability.

By integrating adaptive planning with workflow-driven execution, SmartAuditFlow sets a new standard for LLM-powered smart contract security auditing, enabling more precise, efficient, and scalable vulnerability detection.

The rest of the paper is organized as follows. Section 2 presents the current challenges in smart contract analysis and reviews existing methods. Section 3 provides an in-depth explanation of SmartAuditFlow's architecture and operational mechanisms. Section 4 presents a comprehensive evaluation of SmartAuditFlow, including the datasets used, evaluation criteria, experimental results. Section 5 discusses related works, the findings and examines potential threats to validity. Section 6 concludes the paper by summarizing its contributions and offering insights into the future of smart contract analysis tools.

2 Backgrounds

2.1 Smart Contract Auditing

The rapid growth of smart contracts across blockchain networks has brought transformative capabilities, particularly to the Decentralized Finance (DeFi) sector. This rapid adoption, however, has simultaneously highlighted the critical importance of robust security measures, as vulnerabilities within smart contracts have frequently led to substantial

financial losses [32, 37]. The DeFi market, for instance, reached a peak total value locked (TVL) of approximately $179 billion USD on November 9, 2021, with the Ethereum network alone contributing $108.9 billion USD (around 60% of the total DeFi TVL at the time). As these blockchain-based applications attract significant capital and user engagement, they have also become prime targets for malicious actors. Until the end of 2024, the total amount of money lost by blockchain hackers exceeds 35.32 billion USD, stemming from more than 1800 hacks. A particularly severe recent incident occurred in February 2025, involving the theft of approximately **$1.5 billion** USD in Ethereum from the Bybit digital asset exchange, reportedly one of the largest single cryptocurrency heists to date[2].

A smart contract audit is a systematic security assessment designed to identify and mitigate vulnerabilities, logical errors, and inefficiencies within the contract's codebase. Different from foundational blockchain security concerning consensus mechanisms or virtual machines, smart contract security operates at the application layer, which may be more irregular and complex. During an audit, security experts scrutinize all pertinent materials, including source code, whitepapers, and architectural documentation. This process typically involves manual inspection to uncover nuanced flaws that automated tools might miss, complemented by the use of specialized static and dynamic analysis tools to detect common vulnerability patterns and deviations from best practices [26, 39]. The cost of such audits can range from **$5,000** to over **$50,000** USD, contingent on contract complexity, and demand deep expertise in software engineering, blockchain-specific programming languages (like Solidity), and DeFi protocols. Audit durations can vary from a few days for simple contracts to several weeks or months for complex decentralized applications (dApps).

2.2 Automated Vulnerability Detection

Traditional automated techniques for smart contract vulnerability detection, including static analysis, symbolic execution, and fuzz testing, have been instrumental in early efforts to secure smart contracts [5, 10]. However, these methods face significant challenges related to scalability, computational efficiency, and their ability to comprehensively analyze increasingly complex smart contract systems. A critical study by Zhang et al. [43] revealed that 37 widely-used automated detection tools failed to identify over 80% of 516 exploitable vulnerabilities found in 167 real-world smart contracts, underscoring substantial blind spots in conventional approaches. The study also highlighted that the intricacies of detecting complex bugs often necessitate the collaborative efforts of multiple human auditors, further emphasizing the limitations of standalone automated tools and the high cost of thorough manual reviews.

Recent advancements in LLMs offer a promising frontier for enhancing smart contract security analysis [9, 37]. LLMs demonstrate strong capabilities in understanding and reasoning about both natural language and programming code. This allows them to potentially bridge the gap between intended contract behavior and actual implementation logic, identifying vulnerabilities, verifying compliance with specifications, and assessing logical correctness with a degree of nuance previously unattainable by purely algorithmic tools. LLMs can excel at reasoning about complex security scenarios, explaining identified vulnerabilities in an understandable manner, evaluating business logic intricacies, and even hypothesizing novel attack vectors [6, 30]. Studies exploring LLMs in the smart contract domain have reported promising results, in some instances demonstrating performance comparable or superior to traditional automated tools. However, the probabilistic nature of LLMs means that their outputs can be inconsistent, and developing robust mechanisms for validating their findings remains an active area of research and a critical consideration for practical deployment [2, 7].

[2]https://www.csis.org/analysis/bybit-heist-and-future-us-crypto-regulation

2.3 Agentic Systems and Workflows

Effectively harnessing LLMs for complex tasks like smart contract auditing requires sophisticated interaction strategies beyond simple querying. Prompt engineering, the art of crafting effective inputs to guide LLM outputs, is a cornerstone of LLM application. It is often more computationally efficient and cost-effective for adapting pre-trained LLMs to specific tasks than full model retraining or extensive fine-tuning, especially for rapid prototyping and iteration. However, LLMs are highly sensitive to prompt phrasing; minor variations in input can lead to significantly different, sometimes degraded, performance. For instance, one study demonstrated accuracy drops of up to **65%** across several advanced LLMs due to subtle changes in question framing [27]. To address this sensitivity and enhance LLM reasoning, researchers have developed advanced prompting methodologies. These include techniques like chain-of-thought (CoT) prompting, which encourages step-by-step reasoning [18]; few-shot learning, which provides examples within the prompt [35]; and more complex structured reasoning frameworks like Tree of Thought (ToT) [40] and Graph of Thought (GoT) [4]. These methods aim to elicit more reliable and accurate responses by structuring the LLM's generation process.

Beyond one attempt advanced prompting, more sophisticated operational paradigms are emerging to tackle multi-step, complex problems. Two prominent approaches are autonomous agents and agentic workflows. **Autonomous agents** are designed to perceive their environment, make decisions, and take actions dynamically to achieve goals, often involving flexible, self-directed reasoning and tool use [46]. They typically require careful design of their action space and decision-making heuristics tailored to their operational context. In contrast, **agentic workflows** structure problem-solving by decomposing a complex task into a sequence of predefined or dynamically chosen sub-tasks, often involving multiple LLM invocations, potentially with specialized roles or prompts for each step [13, 42]. This paradigm emphasizes a more explicit orchestration of LLM capabilities, leveraging human domain expertise in designing the workflow structure and allowing for iterative refinement of both the process and the intermediate results [15]. Agentic workflows can integrate external tools, knowledge bases, and feedback loops, making them well-suited for tasks requiring methodical execution and high reliability, such as security auditing.

3 Methodology

In this section, we present the design and operational principles of SmartAuditFlow, our LLM-powered framework for automated smart contract auditing. We begin with an overview of the framework's architecture and its core Plan-Execute reasoning paradigm (Section 3.1). Subsequently, we will discuss in detail the primary operational phases: the Planning Phase, encompassing initial analysis and audit strategy generation (detailed in Section 3.2), and the Execution Phase, covering multi-faceted vulnerability assessment and synthesis (detailed in Section 3.3). Finally, we design a LLM-Powered Audit Evaluator to measure the quality of the audit report (Section 3.4).

3.1 Overview

To address the inherent complexities of smart contract auditing, the uncertainties in LLM-based code analysis, and the significant workload of human auditors, SmartAuditFlow employs a structured **Plan-Execute Reasoning** paradigm. This paradigm decomposes the audit into five sequential stages, each driven by an LLM and tailored to a distinct aspect of the audit, ensuring a systematic, coherent, and actionable review. The overarching aim is to analyze smart contracts and generate a comprehensive, coherent, and actionable audit report.

Figure 1 depicts the high-level architecture of SmartAuditFlow, which is composed of five main components. The *Smart Contract Auditing Workflow* serves as the central processing pipeline and its operational logic is formally defined

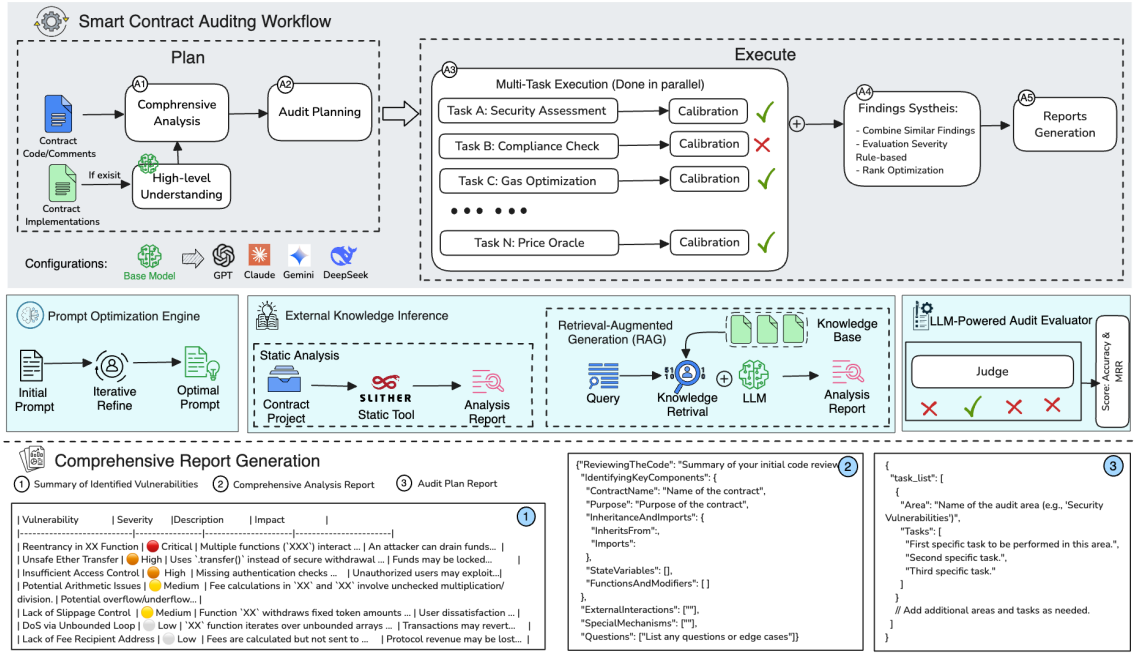


Fig. 1. Overview of the SmartAuditFlow system.

in Algorithm 1. Supporting this workflow, the *Prompt Optimization Engine* is dedicated to the generation, refinement, and validation of optimal prompts, which are crucial for guiding the LLM effectively at each stage. Furthermore, *External Knowledge Inference* augments the LLM's inherent knowledge and improves analytical accuracy by incorporating external data sources such as up-to-date vulnerability databases and security best practices. To ensure the reliability of the audit, the *LLM-Powered Audit Evaluator* establishes a framework for fair, consistent, and reproducible evaluations of the audit outputs generated by the LLM judge. Finally, the *Comprehensive Report Generation* component is responsible for consolidating all validated findings, analyses, and supporting evidence from the Core Auditing Workflow into a structured, actionable, and clear audit report suitable for various stakeholders.

The heart of SmartAuditFlow's operation is in its Core *Auditing Workflow*, which is precisely delineated in Algorithm 1. This algorithm outlines a dynamic, multi-stage process for auditing smart contracts. It employs an LLM ($\mathcal{M}$), guided by a set of specialized prompts ($\mathcal{P}$), takes smart contract code (c) and contextual documents (D_{ctx}) as primary inputs, and produces a comprehensive audit report (R).

Initially, the LLM leverages prompt $\mathcal{P}_{A1}$ with c and D_{ctx} to generate an initial contextual understanding (s_1). Subsequently, this understanding s_1 informs the LLM, via prompt $\mathcal{P}_{A2}$ and code c, in creating a prioritized audit plan (s_2), which is then decomposed into specific sub-tasks ($\mathbf{t}$). Each sub-task t^j then undergoes a multi-faceted execution and validation process; here, the LLM, using task-specific execution ($\mathcal{P}^j_{A3e}$) and validation ($\mathcal{P}^j_{A3v}$) prompts, first performs a preliminary assessment (e^j) and then validates this to yield a finding v^j with a confidence score, aggregating only those findings that meet a predefined threshold into the set of validated findings (s_3). These validated findings s_3 are then subjected to a cross-cutting synthesis, producing a synthesized security posture overview (s_4). The entire workflow

Algorithm 1 Dynamic Smart Contract Auditing Workflow

Require: Smart contract code c, Contextual documents D_{ctx}, LLM $\mathcal{M}$,
Set of specialized prompts $\mathcal{P} = \{\mathcal{P}_{A1}, \mathcal{P}_{A2}, \{\mathcal{P}^i_{A3e}\}, \{\mathcal{P}^i_{A3v}\}, \mathcal{P}_{A4}\}$
Ensure: Comprehensive audit report R

// A1: Context-Aware Initial Analysis
$s_1 \leftarrow \mathcal{M}(\mathcal{P}_{A1}(c, D_{ctx}))$ ▷ Generate initial analysis and understanding
// A2: Adaptive Audit Planning
$s_2 \leftarrow \mathcal{M}(\mathcal{P}_{A2}(s_1, c))$ ▷ Create a prioritized audit plan (list of sub-tasks $\mathbf{t}$)
$\mathbf{t} = \{t^1, t^2, \ldots, t^n\} \leftarrow$ extract_subtasks(s_2)
// A3: Multi-faceted Vulnerability Execution and Validation
$S_v = \emptyset$ ▷ Initialize set of validated findings for s_3
for each sub-task $t^j \in \mathbf{t}$ **do**
  $e^j \leftarrow \mathcal{M}(\mathcal{P}^j_{A3e}(t^j, c, s_1))$ ▷ Preliminary execution for sub-task t^j
  $(v^j, \text{confidence}^j) \leftarrow \mathcal{M}(\mathcal{P}^j_{A3v}(e^j, c))$ ▷ Validate e^j; output validated finding v^j and confidence
  if confidence$^j \geq$ THRESHOLD_CONFIDENCE **then**
    $S_v \leftarrow S_v \cup \{v^j\}$
  end if
end for
$s_3 \leftarrow S_v$ ▷ Aggregated validated findings from execution stage
// A4: Cross-Cutting Findings Synthesis
$s_4 \leftarrow \mathcal{M}(\mathcal{P}_{A4}(s_3))$ ▷ Correlate, prioritize, assign severity to findings in s_3
// A5: Comprehensive Report Generation
$s_5 \leftarrow \{s_1, s_2, s_4\}$
return $R = s_5$

culminates in the generation of the final audit report (R), which contains the initial understanding s_1, the audit plan s_2, and the synthesized findings s_4.

3.2 Plan Phase

The Plan phase is critical for preparing focused and effective tasks for the subsequent Execute Phase. Central to this is guiding the LLM using precisely engineered prompts. Prompts fundamentally shape the LLM's interpretation of smart contract structures, identification of high-risk functions, and overall focus during analysis. As highlighted by prior research, poorly constructed or misaligned prompts can significantly degrade LLM output quality, potentially leading to incomplete audits or misinterpretation of vulnerabilities [44]. Therefore, the systematic generation and selection of optimal prompts for each reasoning step is crucial.

3.2.1 Initial Analysis and Adaptive Planning Stages (A1-A2). The Plan phase comprises two foundational stages: Context-Aware Initial Analysis (A1) and Adaptive Audit Planning (A2). The audit starts with a thorough understanding s_1 of the smart contract project. The objectives at this stage are to interpret the contract's business logic, identify its main roles (e.g., token standard, governance, upgradeability, DeFi interactions), delineate key functionalities, and recognize potentially exposed assets (e.g., funds, sensitive data, privileged permissions). For instance, understanding the design standards for specific token types (e.g., ERC-20, ERC-721) and their common implementation patterns is essential. The LLM systematically analyzes the contract's structure and logic, methodically examining functions and data structures to highlight preliminary areas of interest or potential high-level concerns.

In Stage A2, the LLM leverages insights from the initial analysis (s_1) to formulate a tailored audit plan (s_2). This plan outlines a structured set of sub-tasks, each targeting specific security aspects of the contract, such as reentrancy, access control, or token issue. The plan is designed to focus analytical efforts on the most critical areas first. The quality of

the audit plan depends heavily on the LLM's interpretation of s_1. Additionally, the LLM may be prompted to identify complex functions or contract sections, recommending further decomposition for granular analysis in subsequent stages. This adaptive process ensures the audit strategy is customized for each contract's unique risk profile.

To effectively realize Stages A1 and A2, we employ an iterative prompt optimization strategy inspired by recent advancements in automatic prompt engineering [8, 38, 45]. This strategy aims to discover an optimal instruction ρ^* that, when prepended to a given smart contract code c, maximizes the quality of the LLM's output A (e.g., a comprehensive initial analysis or a robust audit plan). Moreover, Stage A1 is enhanced by integrating insights from static analysis tools to ground the LLM's assessments and improve the accuracy of its initial findings. These insights inform the subsequent prompt generation and planning in later stages.

3.2.2 Iterative Prompt Optimization Framework. In the Plan Phase, each prompt $\mathcal{P}$ is composed of an instruction component ρ and the smart contract code c. Given that c is fixed for a specific auditing task, our optimization focuses on refining the instruction ρ. Formally, consider a task specified by a dataset $\mathcal{D}_{task} = \{(c, A)\}$ of input/output pairs, where c is a contract and A is the expected result (e.g., an expert-crafted initial analysis or audit plan). These reference answers are collected from reputable websites and are also available in our GitHub repository, as listed in Table 9. Our goal is to find an optimal instruction ρ^* that maximizes a scoring function $f(\rho, c, A)$, which measures the alignment of the LLM's output with A. We formulate this objective as the following optimization problem:

$$\rho^* = \arg\max_{\rho} \left(\frac{1}{|\mathcal{D}_{train}|} \sum_{(c,A) \in \mathcal{D}_{train}} f(\rho, c, A) - \lambda \cdot \text{Complexity}(\rho) \right) \tag{1}$$

where $f(\rho, c, A)$ is the multi-criteria scoring function, Complexity(ρ) measures the complexity of the instructions, λ is a regularization hyperparameter, and the maximization is performed over a held-out validation set $\mathcal{D}_{\text{train}}$.

We employ an optimization algorithm to search for ρ^*, outlined in Algorithm 2. This iterative process is designed to converge towards an instruction that elicits accurate, thorough, and contextually relevant outputs from the LLM $\mathcal{M}$.

- **Initial Population Generation** ($\mathcal{U}_0$): We create a diverse initial population of k candidate instructions $\mathcal{U}_0 = \{\rho_1, \rho_2, \ldots, \rho_k\}$ using stochastic sampling and mutation:
  (1) **Stochastic Sampling**: A meta-prompt (see Appendix A.1) guides the LLM to generate varied candidate instructions based on task descriptions and examples of high-performing cases.
  (2) **Mutation of Seed Prompts**: We apply paraphrasing, keyword substitution, and structural edits to existing instructions, controlled by a temperature parameter τ to balance creativity and determinism.
  (3) **Replay Buffer $\mathcal{B}_{\text{replay}}$**: Initialize $\mathcal{B}_{\text{replay}} = \emptyset$, this buffer will store top instructions from previous generations along with their past fitness scores.
- **Evolution Loop with Mini-Batch Optimization [19, 29]**: Let $\mathcal{U}_t$ denote the instruction set at generation t. Each iteration of the loop ($t = 1 \ldots T$) proceeds as follows:
  (1) **Mini-Batch Sampling**: For generation t, a mini-batch $\mathcal{B}_t \subset \mathcal{D}_{train}$ is sampled, with its size n_t (e.g., $n_t = \lceil 0.1|\mathcal{D}_{train}| \cdot (1 + t/T) \rceil$).
  (2) **Stochastic Fitness Evaluation**: Evaluate each candidate $\rho \in \mathcal{U}_{t-1}$ using:

$$f_t(\rho) = \frac{1}{|\mathcal{B}_t|} \sum_{(c,A) \in \mathcal{B}_t} f(\rho, c, A) + \epsilon \cdot \frac{1}{|\mathcal{B}_{\text{replay}}|} \sum_{(\rho', f') \in \mathcal{B}_{\text{replay}}} \text{sim}(\rho, \rho') \cdot f' \tag{2}$$

Algorithm 2 Iterative Prompt Optimization Algorithm

Require: Training dataset $\mathcal{D}_{train}$, validation set $\mathcal{D}_{val}$;
Max generations $T_{\max}$, population size P_S, elite count k_e, offspring count $k_o = P_S - k_e$;
Replay buffer capacity m, mutation parameters $(\tau_{\max}, \beta)$
Convergence criteria: min fitness improvement δ_{fitness}, stable generations N_{stable}, min diversity $D_{\min}$
Ensure: Optimal instruction ρ^*

Phase 1: Initialization:
Generate initial population $\mathcal{U}_0 = \{\rho_1, \rho_2, \ldots, \rho_{P_S}\}$ using mutation of seed prompts
Initialize replay buffer $\mathcal{B}_{\text{replay}} \leftarrow \emptyset$
Initialize $\bar{f}_0(\rho) \leftarrow \text{EvaluateInitialFitness}(\rho, \mathcal{D}_{\text{train}})$ for each $\rho \in \mathcal{U}_0$
$t \leftarrow 1$; $generations\_without\_improvement \leftarrow 0$
Phase 2: Evolutionary Loop
while $t \le T_{\max}$ **and** $generations\_without\_improvement < N_{\text{stable}}$ **do**
  Sample mini-batch $\mathcal{B}_t \subset \mathcal{D}_{train}$ of size $n_t = \lceil 0.1|\mathcal{D}_{train}| \cdot (1 + t/T) \rceil$ ▷ Mini-Batch Sampling
  for all $\rho \in \mathcal{U}_{t-1}$ **do**
    if $|\mathcal{B}_{\text{replay}}| > 0$ **then**
      $f_{\text{replay}}(\rho) \leftarrow \epsilon \cdot \frac{1}{|\mathcal{B}_{\text{replay}}|} \sum_{(\rho', f') \in \mathcal{B}_{\text{replay}}} \text{sim}(\rho, \rho') \cdot f'$
    else
      $f_{\text{replay}}(\rho) \leftarrow 0$
    end if
    $f_{\text{batch}}(\rho) \leftarrow \frac{1}{|\mathcal{B}_t|} \sum_{(c,A) \in \mathcal{B}_t} f(\rho, c, A)$
    $f_t(\rho) \leftarrow f_{\text{batch}}(\rho) + f_{\text{replay}}(\rho)$
  end for
  for all $\rho \in \mathcal{U}_{t-1}$ **do**
    $\bar{f}_t(\rho) \leftarrow \alpha \cdot \bar{f}_{t-1}(\rho) + (1 - \alpha) \cdot f_t(\rho)$
  end for
  Select k_e elites $\mathcal{E}_t$ based on $\bar{f}_t(\cdot)$
  Compute mutation temperature: $\tau_t \leftarrow \tau_{\max} \cdot e^{-\beta t}$
  Select parents $\mathcal{P}_{\text{parents}}$ from $\mathcal{U}_{t-1}$ (e.g., using tournament selection, or use $\mathcal{E}_t$)
  $\mathcal{O}_t \leftarrow \emptyset$
  for $i = 1$ **to** k_o **do** ▷ Generate $k_o = P_S - k_e$ offspring
    $\rho_{\text{parent}} \leftarrow \text{SelectParent}(\mathcal{P}_{\text{parents}})$
    $\rho_{\text{offspring}} \leftarrow \text{Mutate}(\rho_{\text{parent}}, \tau_t, \mathcal{B}_t)$ ▷ Guided mutation using $\mathcal{B}_t$
    $\mathcal{O}_t \leftarrow \mathcal{O}_t \cup \{\rho_{\text{offspring}}\}$
    $\bar{f}_{t-1}(\rho_{\text{offspring}}) \leftarrow \bar{f}_{t-1}(\rho_{\text{parent}})$ ▷ Initialize smoothed fitness for new offspring, or use parent's
  end for
  $\mathcal{U}_t \leftarrow \mathcal{E}_t \cup \mathcal{O}_t$ ▷ New population of size $P_S = k_e + k_o$
  Update $\mathcal{B}_{\text{replay}}$ with top-m performers from $\mathcal{U}_t$
  $\bar{f}_{\text{best},t} \leftarrow \max_{\rho \in \mathcal{U}_t} \bar{f}_t(\rho)$
  if $t > 1$ **and** $|\bar{f}_{\text{best},t} - \bar{f}_{\text{best},t-1}| < \delta_{\text{fitness}}$ **then**
    $generations\_without\_improvement \leftarrow generations\_without\_improvement + 1$
  else
    $generations\_without\_improvement \leftarrow 0$
  end if
  $D(\mathcal{U}_t) \leftarrow 1 - \text{MeanPairwiseSimilarity}(\mathcal{U}_t)$
  $t \leftarrow t + 1$
end while
Phase 3: Final Selection
Using Eq. 1 to select ρ^* on a held-out validation set $\mathcal{D}_{\text{val}}$
return ρ^*

where $f(\rho, c, A)$ is the primary scoring function (Eq. 3), $\text{sim}(\rho, \rho')$ measures similarity between instructions (e.g., using sentence embeddings), ϵ is a hyperparameter weighting the replay-based regularization, and f' representing the previously recorded fitness for ρ' stored in $\mathcal{B}_{\text{replay}}$.

(3) **Elite Selection with Momentum**: The top-k_e elite instructions $\mathcal{E}_t$ are selected based on a moving average of their fitness to preserve high-performing traits and reduce noise from mini-batch sampling:

$$\bar{f}_t(\rho) = \alpha \cdot \bar{f}_{t-1}(\rho) + (1 - \alpha) \cdot f_t(\rho),$$

where $\alpha \in [0, 1]$ is the momentum coefficient.

(4) **Offspring Generation with Guided Mutation**: A new set of offspring instructions $\mathcal{O}_t$ is generated by applying mutation operators to selected parents from $\mathcal{E}_t$. The mutation diversity is controlled via exponential decay $\tau_t = \tau_{\text{initial}} \cdot e^{-\beta t}$. Mutations that yield improved scores on the current mini-batch $\mathcal{B}_t$ can be prioritized or reinforced.

(5) **Population Update**: The next generation's population $\mathcal{U}_t$ is formed by combining elites and offspring, e.g., $\mathcal{U}_t = \mathcal{E}_t \cup \mathcal{O}_t$. The replay buffer $\mathcal{B}_{\text{replay}}$ is updated with the top-performing instructions from $\mathcal{U}_t$.

(6) **Convergence Check**: The loop terminates if the improvement in the moving average fitness of the top elites plateaus (e.g., $\Delta\bar{f}_t < \delta$ for several generations) or if population diversity $D(\mathcal{U}_t) = \frac{1}{k^2} \sum_{\rho_i, \rho_j \in \mathcal{U}_t} \text{sim}(\rho_i, \rho_j)$ falls below a threshold.

- **Final Instruction Selection**: Upon termination of the evolutionary loop, the instruction ρ^* from the final population that maximizes the objective function defined in Eq. 1 (evaluated on the held-out validation set $\mathcal{D}_{\text{val}}$) is selected as the optimal instruction for the given task.

Multi-Criteria Scoring Function $f(\rho, c, A)$. A simplistic scoring function (e.g., binary success/failure) may fail to capture nuanced alignment between model outputs and desired results. Therefore, we combine multiple criteria (e.g., correctness, efficiency, and user satisfaction) into $f(\rho, c, A)$:

$$f(\rho, c, A) = w_{\text{exec}} \cdot f_{\text{exec}}(\mathcal{M}(\rho, c), A) + w_{\text{log}} \cdot f_{\text{log}}(\mathcal{M}(\rho, c), A) \tag{3}$$

where $\mathcal{M}(\rho, c)$ is the output generated by the LLM when prompted with instruction ρ and contract c. The weights w_{exec} and w_{log} (e.g., $w_{\text{exec}} + w_{\text{log}} = 1$) balance the contribution of each component.

Output Alignment Score (f_{exec}): This component measures how well the LLM's generated output $\mathcal{M}(\rho, c)$ aligns with the expected output A from the dataset $\mathcal{D}_{task}$. For smart contract auditing tasks, A might represent an expert-identified set of vulnerabilities, a correctly summarized contract behavior, or a well-structured audit plan component. Alignment can be measured using metrics like BERTScore, ROUGE, or BLEU [11, 33] for textual similarity if A is text, or more domain-specific metrics. We define a domain-specific accuracy metric, drawing upon best practices and guidelines from leading smart contract audit firms (e.g., ConsenSys Diligence, Trail of Bits, or OpenZeppelin).

$$f_{\text{exec}}(\mathcal{M}(\rho, c)) = w_{\text{cov}} \cdot f_{\text{coverage}}(\mathcal{M}(\rho, c), A) + w_{\text{det}} \cdot f_{\text{detail}}(\mathcal{M}(\rho, c), A) \tag{4}$$

where $f_{coverage}$ ensures the output $\mathcal{M}(\rho, c)$ identifies all relevant structural components, and f_{detail} assess the accuracy and completeness for each component.

Output Likelihood Score (f_{log}): This component uses the negative log-likelihood assigned by the LLM $\mathcal{M}$ to its own generated output. A higher log probability suggests that the output is more confident according to the LLM's internal

model. It is typically calculated as the average negative log-likelihood per token of the output sequence $O = w_1, \ldots, w_N$ generated by $\mathcal{M}(\rho, c)$:

$$\mathcal{f}_{\log}(\mathcal{M}(\rho, c), A) = -\frac{1}{N} \sum_{i=1}^{N} \log P_{\mathcal{M}}(w_i | \rho, c, w_1, \ldots, w_{i-1}) \tag{5}$$

Hyperparameter Tuning Strategy. To ensure reproducibility and performance consistency, we use the following settings to tune the Algorithm 2's key parameters:

- *Evolutionary hyperparameters:* population size P_S=20, elites k_e=4, offspring k_o=16; replay buffer capacity m=50; similarity sim$(\cdot)$= cosine over Sentence-BERT embeddings (`all-mpnet-base-v2`); momentum α=0.6; replay weight ϵ=0.1; diversity threshold D_{min}=0.35; convergence δ=0.002 with N_{stable}=5 generations.
- *Mutation schedule:* τ_{max}=0.9 with exponential decay β=0.05; if $D(\mathcal{U}_t) < D_{\text{min}}$ for three consecutive generations, we temporarily increase τ_t by 15% to encourage exploration.
- *Mini-batch schedule:* $n_t = \lceil 0.1 |\mathcal{D}_{train}| (1 + t/T) \rceil$; total generations T=30.
- *LLM decoding (for instruction search runs):* temperature 0.2, top-p 0.95, max tokens 4096.
- *Tuning protocol:* We performed a grid search on a held-out subset of 50 contracts from the Prompt Optimization Set, maximizing Eq. 3 while minimizing Complexity(ρ). Explored ranges included $P_S \in \{16, 20, 32\}$, $k_e \in \{4, 5, 6\}$, $\tau_{\text{max}} \in \{0.7, 0.9, 1.1\}$, $\beta \in \{0.03, 0.05, 0.08\}$, $m \in \{32, 50, 64\}$, and $D_{\text{min}} \in \{0.30, 0.35, 0.40\}$. Random seed was fixed (42) for all sweeps.

Representative optimal prompts for Stages A1/A2 are provided in Appendix A.4 for direct reuse. Additional implementation details (config files, seeds, and logs) are documented in Appendix A.

3.2.3 Static Analysis Tool Involvement. While LLMs demonstrate remarkable capabilities in processing natural language and understanding high-level code logic, they may not consistently identify certain edge-case scenarios or subtle security flaws rooted in complex code interdependencies or low-level implementation details. To address this and enrich the initial stages of our audit workflow, we proposed to incorporate insights from static analysis tools.

Widely recognized tools such as Slither, Mythril, Solint, and Oyente automatically detect common vulnerability patterns. However, their utility in our framework extends beyond standalone detection. Specifically, within Stage A1, the outputs from these tools are leveraged to significantly enhance the LLM's judgements. Among these tools, we selected slither-0.11.0 as our static analysis tool. This choice is motivated by Slither's active maintenance, frequent updates driven by a large developer community, and its comprehensive suite of nearly 100 distinct vulnerability detectors. In contrast, tools like Mythril primarily focus on low-level EVM bytecode analysis, which, while valuable, offers a different granularity of insight, and other tools such as Solint and Oyente have seen limited updates in recent years. Slither's proficiency in extracting detailed code syntax and semantic information (e.g., inheritance graphs, function call graphs, state machine representations) has been well-documented in prior research [1, 14].

Furthermore, the integration of static analysis serves as a crucial cross-referencing mechanism. By comparing LLM-generated insights about potential vulnerabilities against the findings from static analysis, SmartAuditFlow can mitigate the risk of LLM "hallucinations". This hybrid approach effectively balances automated pattern detection with contextual reasoning.

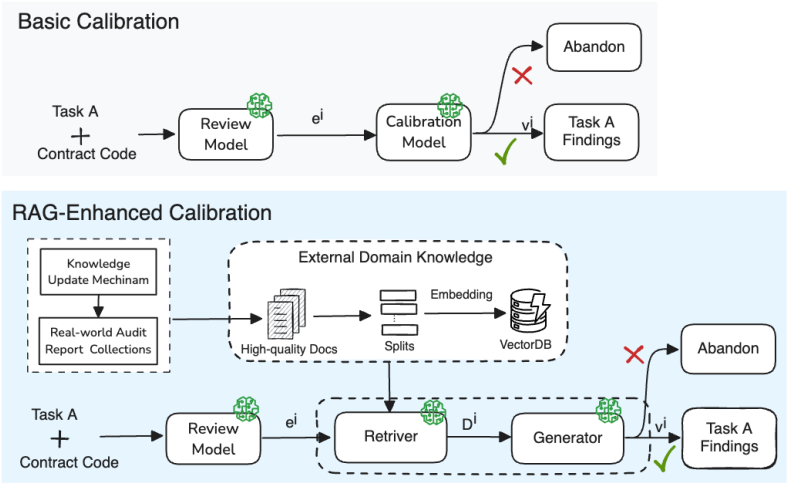


Fig. 2. (a) Basic Calibration directly uses LLM to validate the initial review finding; (b) RAG-Enhanced Calibration integrates external knowledge for robust validation and comprises two primary components: Retriever and Generator.

3.3 Execute Phase

The Execute Phase operationalizes the adaptive audit plan by performing detailed, sub-task-specific security checks on the smart contract. Unlike the Plan Phase, where prompts are often more extensive to define strategic tasks or perform broad initial analyses, the prompts utilized for individual sub-tasks in the Execute Phase are typically simpler and more direct. Each sub-task is narrowly focused on a single security aspect (e.g., checking for reentrancy in a specific function, evaluating access controls for a particular state variable). This focused approach guides the LLM to conduct a deep exploration of a well-defined issue.

3.3.1 Generating and Delivering Validated Audit Findings (A3-A5). The Execute Phase begins with the Multi-Task Execution (A3), where the framework executes the prioritized sub-tasks defined in the audit plan (s_2). For each sub-task (e.g., *check function X for reentrancy*, *analyze access controls for admin functions*), the LLM performs a detailed security assessment on the relevant code segments. This generates preliminary findings. Crucially, these preliminary findings undergo a **calibration process**. This calibration involves the LLM re-evaluating its own initial finding, potentially guided by additional information such as rule-based checks or specific validations. Only findings s_3 that meet predefined confidence thresholds or validation criteria are retained for synthesis. Retrieval-Augmented Generation (RAG) techniques may be optionally employed here to provide the LLM with access to up-to-date vulnerability databases or best practice guidelines during both assessment and validation, further enhancing accuracy.

Validated findings from the execution stage (s_3) are aggregated and synthesized to build a coherent and prioritized overview of the contract's security posture (s_4). This involves more than just listing vulnerabilities. It assigns severity levels to each finding based on factors like potential business impact (e.g., financial loss, loss of control, denial of service) and exploitability (e.g., conditions required, attacker privileges needed, public knowledge of exploit). The synthesis process also aims to identify relationships between vulnerabilities, such as shared root causes or dependencies. This systematic classification and correlation allow the system to filter and present findings in a manner that highlights the most urgent issues requiring attention.

The final stage focuses on producing a clear, concise, and actionable audit report. This report consolidates the initial contract understanding (s_1), the audit plan (s_2), and the synthesized findings (s_4), including severity assessments and, where possible, actionable recommendations for mitigation. The structure and language of the report can be tailored to suit different audiences (e.g., developers requiring technical details, or management needing high-level summaries). Drawing insights from best practices in security reporting and relevant studies [23, 41], the report aims to cover key dimensions such as clear vulnerability descriptions, severity evaluations, precise localization of issues in the code, and concise summaries.

3.3.2 Task-Specific Review and Calibration. Given that LLMs can occasionally "hallucinate" or generate outputs that deviate from factual contract code [3]. Stage A3 is explicitly structured into two critical steps for each sub-task: (1) initial task-specific review and (2) rigorous calibration. This two-step process is designed to maximize accuracy and minimize false positives or missed vulnerabilities before findings are aggregated.

(1) Task-Specific Review (e^i). In this initial step, the LLM is prompted to investigate one narrowly defined security concern as dictated by the sub-task t^i from the audit plan. The objective is to have the LLM scrutinize specific code segments or contract behaviors relevant to that concern. Example prompts for this stage might include:

- *Analyze function* `withdrawFunds()` *for potential reentrancy vulnerabilities. Detail any observed patterns and the conditions under which they might be exploitable.*
- *Examine all division operations within this contract for susceptibility to integer overflow or underflow issues. List each susceptible operation and its location.*

By focusing on a singular security issue, the LLM's analysis remains targeted, leveraging its understanding of contract logic and known security best practices to pinpoint potential anomalies. The output of this step for each sub-task t^i is an initial review finding e^i, formally expressed as:

$$e^i = \mathcal{M}(\mathcal{P}_e^i(t^i, c)) \tag{6}$$

where $\mathcal{P}_e^i$ is the task-specific review instruction for sub-task t^i, c is the relevant smart contract code (or snippets thereof), and $\mathcal{M}$ represents the LLM.

(2) Calibration v^i. Following the initial review, the calibration step aims to validate, refine, and expand the preliminary findings e^i. This involves guiding the LLM to critically re-examine its initial assessment. The LLM is challenged to ground its claims firmly in the contract's code and relevant security principles. An example calibration prompt could be:

Explain how the identified vulnerability in e^i can be exploited based on the contract's actual variables and functions. Provide a code snippet highlighting the relevant lines. Also describe any defenses that might already exist in the code.

This self-correction and evidence-gathering process helps to filter out unsupported assertions and reduce LLM hallucinations. The output is a calibrated finding v^i:

$$v^i = \mathcal{M}(\mathcal{P}_v^i(e^i, t^i)) \tag{7}$$

where $\mathcal{P}_v^i$ is the calibration instruction, which uses the initial review e^i, the original sub-task context t^i.

3.3.3 RAG-Enhanced Calibration for Improved Accuracy and Grounding. While pre-trained LLMs possess extensive general knowledge, their understanding of domain-specific knowledge can be limited or outdated. This can lead to gaps in analysis or less precise explanations, particularly for edge-case scenarios. To mitigate this, we enhance the calibration

step with Retrieval-Augmented Generation (RAG) [17], as illustrated in Figure 2. RAG augments the LLM's internal knowledge by providing access to relevant excerpts from external, up-to-date knowledge sources during generation.

We have observed instances where an LLM might correctly identify a vulnerable code location but provide an incomplete or imprecise explanation, especially for complex or uncommon vulnerabilities. RAG addresses this by retrieving documents that offer detailed, authoritative explanations for similar issues. Within our Execute Phase, each calibration prompt $\mathcal{P}_v^i$ is augmented with information retrieved from a curated knowledge base. This knowledge base includes smart contract databases (e.g., SWC Registry, CWE, CVE), security guidelines (e.g., Ethereum Foundation, ConsenSys Diligence), seminal research papers, technical blogs detailing novel exploits, and community-vetted best practice repositories (see Appendix B for a list of sources). These resources are embedded and stored in a Vector Database (using models like *Sentence-BERT* [28]) for efficient semantic retrieval.

The RAG-enhanced calibration workflow proceeds as follows:

(1) Formulate Retrieval Query (Q_v^i): Based on the sub-task t^i and the initial review output e^i, a targeted retrieval query is formulated. This query typically includes keywords related to the potential vulnerability (e.g., "integer division", "reentrancy", "`call.value` usage"), function names, and other contextual cues from t^i and e^i.
(2) Retrieve Relevant Documents (D_v^i): The system queries the VectorDB using Q_v^i to retrieve the top-k most relevant document snippets $D_v^i = \{d_1, d_2, \ldots, d_k\}$.
(3) Evidence-Grounded Calibration: The LLM then synthesizes the information from these retrieved documents D_v^i with the initial review output e^i and the original sub-task context t^i to produce the calibrated finding v^i. The calibration instruction $\mathcal{P}_v^i$ is structured to explicitly instruct the LLM to incorporate and reference the retrieved evidence. This is formally represented as:

$$v^i = \mathcal{M}\big(\mathcal{P}_v^i(e^i, t^i, c, D_v^i)\big), \text{ where } D_v^i = \text{Retrieve}(Q_v^i) \tag{8}$$

By compelling the LLM to ground its calibration in authoritative, domain-specific references, we significantly enhance the accuracy, reliability, and explanatory depth of the findings. Each calibrated finding v^i that emerges from this RAG-enhanced process is thus considered a robust and well-substantiated observation, ready for aggregation in Stage A4. Additional details, including further instances and sources for RAG design, are provided in Appendix B

The final output of the Execute Phase (Stage A3) is a list of *validated findings* (v^i), each substantiated by evidence from the contract code and, where applicable, external knowledge sources via RAG. These high-confidence findings are then passed to Stage A4 (Findings Synthesis), where they are further analyzed, prioritized by severity, and checked for inter-dependencies.

3.4 LLM-Powered Audit Evaluator

Assessing the quality of the generation is an even more arduous task than the generation itself, and this issue has not been given adequate consideration. Few studies have pointed that LLM has been a good evaluator for natural language generation (NLG), and the existing approaches are not tailored to the specific needs of the audit process [12, 24]. Inspired by RAGAs [11], we designed and implemented an automated LLM-Powered Audit Evaluator. This evaluator systematically compare the findings generated by SmartAuditFlow against expert-annotated ground truth answers from the same expert audited-report. The primary objectives of this evaluator are to ensure fair and reproducible assessments, accelerate the evaluation cycle, and provide actionable insights for the continual improvement of the SmartAuditFlow

framework. Our LLM-Powered Audit Evaluator operates in two main phases: (1) LLM-Driven Finding Comparison and (2) Quantitative Performance Scoring.

The evaluator requires two structured inputs for each smart contract assessed: (i) the **SmartAuditFlow-generated report**, detailing each detected vulnerability with its type, natural language description, specific code location(s), and assigned severity; and (ii) the corresponding **Standard Answer report**. Standard Answers are curated from existing human-expert audit reports for the identical smart contract version, transformed into the same structured format as the SmartAuditFlow output to facilitate direct comparison. This ground truth includes expert-validated vulnerability types, descriptions, locations, and severities.

LLM-Driven Finding Comparison. This phase focuses on accurately mapping each finding reported by SmartAuditFlow to its corresponding entry in the Standard Answer, or identifying it as unique. An LLM is central to this process, leveraging its semantic understanding capabilities to interpret nuanced textual descriptions and contextual information. The key steps include:

- **Finding Pairing:** For each vulnerability reported by SmartAuditFlow, the LLM identifies the most plausible candidate match(es) from the Standard Answer. This step also considers findings in the Standard Answer to identify those potentially missed by SmartAuditFlow.
- **Multi-Dimensional Semantic Assessment:** Each candidate pair (one finding from SmartAuditFlow, one from the Standard Answer) is subjected to a detailed comparative analysis by an LLM. This assessment uses a predefined rubric considering multiple dimensions: (a) vulnerability type; (b) description semantic similarity (assessing if both descriptions refer to the same underlying issue); and (c) code location precision (e.g., exact line match, function-level match, overlapping blocks).
- **Match Categorization:** Based on the multi-dimensional assessment and the rubric, the LLM categorizes the relationship between the paired findings. Inspired by prior research [36], we adopt a nuanced classification approach rather than a binary true/false distinction. The primary categories include `Exact Match` (all critical aspects align), `Partial Match` (core issue identified but differs on a specific dimension like severity or location precision), and `Incorrect` (the response misidentifies the core issue in the Standard Answer).

Quantitative Performance Scoring. This phase translates the categorized outputs from the comparison phase into quantitative metrics to evaluate SmartAuditFlow's overall performance. For metrics like accuracy, MRR and MAP, findings in the SmartAuditFlow report are assumed to be ranked (e.g., by severity or an internal confidence score). These metrics are described in Section 4.2.2.

4 Evaluation

This section presents the evaluation of our proposed framework. We assess its performance in detecting smart contract vulnerabilities and compare it against established tools and other LLM-based methods.

4.1 Research Questions

Our evaluation is guided by the following research questions (RQs):

- **RQ1: How effectively does SmartAuditFlow identify common vulnerability types?** This question investigates the models' ability to detect known vulnerabilities as defined in the SWC-registry. We compare the performance of our framework against other methods in identifying common vulnerabilities.

- **RQ2: How does SmartAuditFlow perform when evaluated using different detection metrics?** We assess the performance of SmartAuditFlow by considering vulnerabilities identified within the top-N ranked results and conventional detection metrics. This investigates the framework's ability to prioritize critical issues effectively.
- **RQ3: How does SmartAuditFlow perform on complex real-world projects?** Real-world smart contracts often exhibit greater complexity than curated benchmark datasets. We evaluate SmartAuditFlow on two datasets of audited real-world projects and compare its performance to existing tools.
- **RQ4 What is the impact of different underlying LLMs on SmartAuditFlow's performance?** With the rapid evolution of LLMs, we investigate how the choice of the LLM backbone (e.g., models from the GPT, Claude, DeepSeek, Gemini series) affects SmartAuditFlow's vulnerability detection accuracy and overall effectiveness.
- **RQ5: How does SmartAuditFlow compare to other contemporary LLM-based vulnerability detection methods?** We evaluate our method on a relevant dataset (the CVE set), comparing its ability to correctly identify vulnerabilities and generate comprehensive, accurate descriptions against other LLM-based approaches.
- **RQ6: How does the integration of different external knowledge sources influence SmartAuditFlow's performance?** We conduct an ablation study to assess the impact of integrating static tool analysis and RAG on the overall performance of SmartAuditFlow, compared to a baseline version without these enhancements.

4.2 Experimental Setup

4.2.1 Dataset. To rigorously evaluate SmartAuditFlow, we utilize a diverse set of datasets that reflect various scenarios and complexities in smart contract auditing. These include:

- **Prompt Optimization Set**: This dataset comprises 1,000 smart contracts from various domains (e.g., DeFi, NFTs, gaming), selected to encompass a wide range of contract structures and potential security concerns. This set is exclusively used for the iterative optimization of prompt instructions (ρ) for Stages A1 and A2 within our Plan Phase, as detailed in Section 3.2. Statistics and construction details for this dataset are provided in Appendix A (Table 6).
- **Standard Vulnerability Set**: This dataset contains smart contracts annotated with common vulnerability types, which serves as a benchmark for evaluating the detection of well-understood vulnerability types against established tools. We utilize the SmartBugs-curated dataset [10], which is widely adopted by both developers and researchers. The dataset includes 143 contracts, each annotated with common vulnerabilities based on the DASP classification, making it an excellent resource for benchmarking vulnerability detection tools.
- **Code4rena Dataset**: To assess performance on more complex, real-world scenarios, we use a curated subset of contracts from projects audited by Code4rena [21, 43]. This set consists of 72 projects, encompassing 6,454 contracts, among which 243 contracts have known issues, including 784 instances of high or medium-severity vulnerabilities. This dataset tests SmartAuditFlow's capability to handle complex contract logic and diverse vulnerability manifestations.
- **Sherlock Dataset:** To improve diversity and mitigate the limitations of relying on a single real-world source, we curated an additional dataset from the Sherlock auditing platform [21]. We adopt a time-span strategy to construct the dataset, selecting projects from a defined period to capture the full spectrum of reported vulnerabilities. The dataset comprises 22 high-profile projects audited between August 2024 and July 2025,

covering 65 contracts and 185 high- or medium-severity vulnerabilities. Each project has publicly available audit reports, which we used to establish precise ground-truth annotations.

- **CVE Set**: This dataset comprises well-known smart contract Common Vulnerabilities and Exposures (CVEs). While there were 595 smart contract CVEs recorded as of July 1, 2025, a large portion of these—particularly older ones—represent similar vulnerability types. For example, 395 of the 595 CVEs are related to integer overflows, with the remaining types distributed as shown in Table I. To enable focused, non-redundant evaluation across diverse security threats, we adopted the curated CVE benchmark from PropertyGPT [25].

All curated datasets, including our annotations, ground truth mappings, and processing scripts, are made openly accessible via our GitHub repository to foster reproducibility and community-driven extensions.

Both Code4rena and Sherlock include real-world projects that present significant challenges even for professional auditing firms. Most standalone static analysis tools, such as Slither, Mythril, and Conkas, failed to detect any vulnerabilities in these datasets—findings that align with prior studies [25, 43]. We also explored other publicly available datasets, such as DeFiVulnLabs[3] and DeFiHackLabs[4]. For DeFiVulnLabs dataset, the GPT-4o model achieving a score of 37/50. DeFiHackLabs, primarily consisting of proof-of-concept exploit pieces of code snippets, was deemed more suitable for attack replication studies than for systematic vulnerability detection benchmarking of full contracts. As a result, we excluded these two datasets from our study.

4.2.2 Evaluation Criteria. The primary objective of our evaluation is to assess the effectiveness of the SmartAuditFlow framework in identifying vulnerabilities in smart contracts. This involves a systematic comparison of SmartAuditFlow's findings against the ground truth for each contract. The classification outcomes are defined as:

- **True Positive (TP)**: The framework correctly identifies a vulnerability that exists in the contract, as validated against the ground truth through our LLM-Powered Audit Evaluator.
- **False Positive (FN)**: The framework fails to detect a vulnerability that is present in the contract according to the ground truth.
- **False Positive (FP)**: The framework reports a vulnerability that does not exist in the ground truth.

Given that SmartAuditFlow, like many LLM-based systems, generates a ranked list of potential vulnerabilities rather than simple binary classifications, relying solely on aggregate counts of TPs, FPs, and FNs can be insufficient for evaluating its practical utility. Our evaluation therefore employs both ranking-based and conventional detection metrics to provide a comprehensive performance profile.

Top-N Accuracy. We assess if a true vulnerability (TP) is identified within the top-N findings reported by the framework (e.g., for N=1, 3, 5). This method is useful, especially in complex tasks like smart contract vulnerability detection, where vulnerabilities may not always be ranked perfectly.

Mean Reciprocal Rank (MRR). This metric evaluates how highly the first correctly identified relevant vulnerability (TP) is ranked for each contract (or query). A higher MRR indicates that the method tends to rank true vulnerabilities closer to the top. The formula is:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \tag{9}$$

[3]https://github.com/SunWeb3Sec/DeFiVulnLabs
[4]https://github.com/SunWeb3Sec/DeFiHackLabs

where $|Q|$ is the total number of queries, and rank_i is the rank of the first TP for the i-th query (or 0 if no TP is found).

Mean Average Precision (MAP). MAP provides a comprehensive measure of both precision and the ranking order of all identified vulnerabilities. It is the mean of Average Precision (AP) scores across all contracts/queries. AP for a single query is calculated as:

$$AP = \frac{1}{N} \sum_{n=1}^{N} P(n) \cdot \text{rel}(n) \tag{10}$$

where $P(n)$ is the precision at rank n (i.e., the fraction of TPs in the top n results), and $\text{rel}(n)$ is the relevance of the n-th result, where 1 indicates a TP and 0 indicates a FN.

Conventional Detection Metrics. Precision (P) measures the proportion of correctly identified vulnerabilities among all reported vulnerabilities: $P = \frac{TP}{TP+FP}$. Recall (R) measures the proportion of correctly identified vulnerabilities among all ground-truth vulnerabilities: $R = \frac{TP}{TP+FN}$. The F1-score is the harmonic mean of precision and recall: $F1 = \frac{2 \cdot P \cdot R}{P+R}$.

4.3 Implementation Details

All experiments for SmartAuditFlow and static tools were conducted within a Docker container to ensure a consistent, isolated, and reproducible environment across all test runs. For evaluating the vulnerability detection capabilities involving LLMs, we adopted a single attempt approach (often denoted as pass@1) [36]. This reflects practical auditing scenarios where auditors typically seek an effective assessment in one primary attempt.

As detailed in Section 3.2, our framework employs an iterative strategy to optimize prompt instructions (ρ) for Stages A1 and A2. For the initial seeding of the evolutionary prompt optimization process (i.e., generating the diverse starting population of candidate prompts), we utilized GPT-4o (gpt-4o-2024-11-20) as an `assistant LLM` to generate an initial set of 20 diverse prompt candidates for both A1 and A2 steps. The Prompt Optimization Set (as described in Section 4.2.1) was partitioned into a 70% subset for running the evolutionary optimization algorithm and a 30% subset for validating the performance of the finally selected optimized prompts. Illustrative examples of both initial and optimized prompts are provided in Appendix A.

For the majority of our experiments evaluating the core capabilities and comparative performance of SmartAuditFlow, we employed GPT-4o as the default LLM integrated within the framework. This model was selected based on its strong performance demonstrated in our preliminary assessments and its advanced reasoning capabilities. We systematically substituted the primary operating LLM within the framework with several other leading models: Gemini-2.5-pro (gemini-2.5-pro-preview-03-25), Claude-3.7-sonnet (claude-3-7-sonnet-20250219), and DeepSeek-v3. These models were accessed via their respective public APIs.

4.4 Experimental Results

To answer the research questions posed earlier, we present the experimental results.

4.4.1 RQ1: Effectiveness in Identifying Common Vulnerabilities. To address RQ1, we evaluated SmartAuditFlow's effectiveness in identifying common vulnerability types using the Standard Vulnerability Set. We benchmarked SmartAuditFlow against a comprehensive suite of baselines, including established traditional static analysis tools and a range of standalone LLMs prompted with a standardized, direct querying strategy. The traditional static analysis tools selected for comparison were Slither (0.11.0), Mythril (0.24.7), and Conkas. For standalone LLM baselines, we

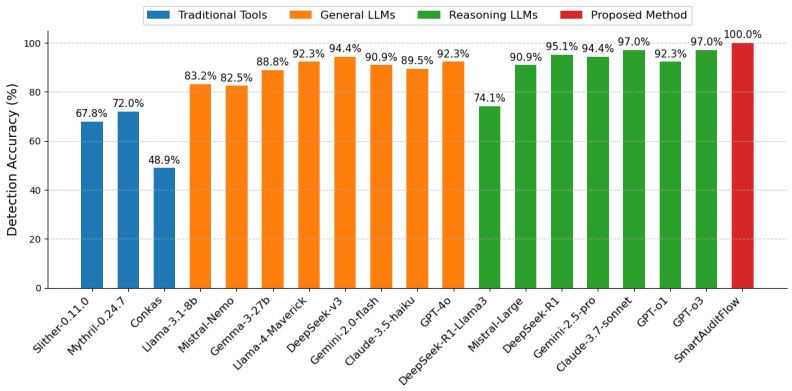


Fig. 3. Evaluation of Smart Contract Vulnerability Detection Tools - A Comparative Analysis

categorized and evaluated two types of models: **General-Purpose Instruction-Tuned Models**: Llama-3.1-8b-it, Mistral-Nemo-Instruct, Gemma-3-27b-Instruct, Llama-4-Maverick-Instruct, DeepSeek-v3, Gemini-2.0-flash, Claude-3.5-haiku (claude-3-5-haiku-20241022), and GPT-4o (gpt-4o-2024-11-20); **Reasoning Models**: DeepSeek-R1-Llama3-8b, Mistral-Large (mistral-large-latest), DeepSeek-R1, Gemini-2.5-pro (gemini-2.5-pro-preview-03-25), Claude-3.7-sonnet (claude-3-7-sonnet-20250219), GPT-o1 (o1-2024-12-17), and GPT-o3 (o3-2025-04-16).

For evaluating the detection capability of all methods, including standalone LLMs, we considered a true vulnerability to be detected if it was correctly identified and reported anywhere in the method's output (effectively a recall-oriented measure for each ground truth instance). The results of this comparative evaluation are summarized in Figure 3.

Our findings indicate that SmartAuditFlow (utilizing its primary configuration with GPT-4o, unless specified otherwise) achieves a detection accuracy of 100% on this dataset. This performance significantly surpasses that of the traditional static analysis tools. For instance, the best-performing traditional tool, Mythril, achieved an accuracy of 72.0%. Notably, even the standalone LLM baselines generally outperformed traditional tools; for example, the lowest-performing LLM in our set, DeepSeek-R1-Llama3-8B, still achieved 74.1% accuracy. This underscores the substantial potential of LLMs in vulnerability detection.

Furthermore, SmartAuditFlow also demonstrated a clear advantage over standalone LLM applications. As an example of its workflow-driven enhancement, when SmartAuditFlow was configured to use GPT-4o as its internal LLM, it achieved the 100% accuracy, representing a 7.7 percentage point improvement over using GPT-4o (pass@1). The results also reveal that reasoning models exhibited superior performance compared to their general-purpose models from the same family or size class. For instance, GPT-o3 (97.0%) and Claude-3.7-Sonnet (97.0%) outperformed models like the baseline GPT-4o (92.3%) and Claude-3.5-Haiku (89.5%). The results also confirmed the general trend that larger-parameter models and those specifically optimized for reasoning or coding tasks in this domain compared to smaller or more generic instruction-tuned models.

Answer to RQ1: SmartAuditFlow demonstrates high effectiveness in identifying common smart contract vulnerabilities, achieving a detection accuracy of 100% on the Standard Vulnerability Set. This significantly outperforms both traditional static analysis tools and standalone applications of advanced LLMs. The results affirm that

strategically applying LLMs within a structured workflow like SmartAuditFlow substantially enhances their intrinsic capabilities for vulnerability detection.

4.4.2 RQ2: Performance Across Different Detection Metrics. While RQ1 assessed overall detection capabilities at top-*max* accuracy, practical auditing often involves reviewing a limited subset of reported findings. Auditors may prioritize inspecting only the top few (e.g., top-1, top-5, or top-10) predictions, especially when dealing with tools that generate a large number of outputs, which can include false positives or less critical information. Therefore, for RQ2, we evaluate how effectively different methods rank true vulnerabilities by examining their performance at various top-N thresholds (top-1, top-5, and top-*max*). We also measure the MRR, the average number of findings generated per contract, and additionally incorporate conventional detection metrics (Precision, Recall, and F1-score) to provide a balanced view of accuracy and robustness.

Table 1 demonstrates our method's superior performance across all top-N accuracy metrics, achieving **66.4% at top-1**, **99.3% at top-5**, and **100% at top-*max***. This indicates that not only does SmartAuditFlow detect a comprehensive set of vulnerabilities, but it also ranks them highly. For comparison, a high-performing standalone LLM baseline, GPT-o3, achieved 58.7% at top-1, 93.0% at top-5, and 97.2% at top-*max*. As expected, for all models, detection accuracy increases with larger N, underscoring the utility of ranked outputs; however, the rate of improvement and the accuracy achieved at lower N values are key differentiators.

The MRR scores further validate SmartAuditFlow's superior ranking capabilities. With an **MRR of 0.8**, SmartAuditFlow ranks the first true positive vulnerability, on average, at approximately position 1.25 (calculated as $1/0.8 = 1.25$). This implies that auditors using SmartAuditFlow would typically encounter the first critical issue within the top one or two reported items. In contrast, other strong standalone LLMs like Claude-3.7-sonnet (with an MRR of 0.73) and GPT-o3 (MRR of 0.72) rank the first true positive at average positions of approximately 1.37 and 1.39, respectively. Models with lower MRR scores, such as DeepSeek-R1-Llama3-8B (MRR of 0.54, corresponding to a top-5 accuracy of 73.4%), require auditors to sift through more findings to locate the first true positive.

The conventional detection metrics reveals that SmartAuditFlow maintains competitive precision (18.1%) despite achieving perfect recall (100%), resulting in the highest F1-score (30.7%) among all LLM-based methods. Notably, some models with comparable recall—such as Claude-3.7-sonnet (R = 97.9%) or GPT-o3 (R = 97.2%)—exhibit lower precision (10.2% and 12.1%, respectively), leading to lower F1-scores than SmartAuditFlow. Traditional static analysis tools like Mythril achieve highest precision (28.0%) but at the cost of significantly lower recall (72.0%), with an average of 2.9 findings per contract often lacking contextual explanations. Slither reports the low precision (14.2%), meaning a substantial portion of its detections are incorrect and, like many rule-based tools, are presented without detailed evidence to aid verification. Conkas exhibits a similar pattern, with moderate precision (20.8%) and a low average output count (2.9), but its results are also typically terse and lack guidance on exploitability. Such limitations in presentation and interpretability increase the burden on experts to manually validate each finding.

Efficiency, in terms of the average number of findings reported per contract, is also a critical factor. Figure 4 illustrates the trade-off between top-*max* accuracy and this output volume. Some models, like DeepSeek-v3 (achieving 94.4% top-*max* accuracy), generate a high number of average outputs (e.g., 15.1 per contract), which improves overall recall (top-*max*) but potentially increases auditor workload due to a higher number of findings to review (including potential false positives). General-purpose instruction-tuned models such as Llama-3.1-8b-it and Mistral-Nemo-Instruct achieved moderate top-*max* accuracy (around 83%) but also produced a relatively high average of 7.7 to 12 findings. Reasoning-enhanced standalone LLMs (e.g., DeepSeek-R1, Claude-3.7-sonnet, and GPT-o3) demonstrated improved top-*max*

Table 1. Comparative Results of SmartAuditFlow and Baselines on the Standard Vulnerability Set

Tool	P(%)	R(%)	F1(%)	top-1(%)	top-5(%)	top-max(%)	MRR	Avg. Outputs
Slither-0.11.0	14.2	67.8	23.5	-	-	-	-	5.1
Mythril-0.24.7	**28.0**	72.0	**40.3**	-	-	-	-	2.9
Conkas	20.8	48.9	29.2	-	-	-	-	2.9
Llama-3.1-8b-Instruct	7.0	83.2	12.9	37.1	79.0	83.2	0.54	12.0
Mistral-Nemo-Instruct	10.7	82.5	19.0	37.1	76.9	82.5	0.51	7.7
Gemma-3-27b-it	17.9	88.8	29.8	51.0	87.4	88.8	0.65	5.0
Llama-4-Maverick-Instruct	11.6	92.3	20.6	38.5	84.6	92.3	0.55	8.0
DeepSeek-v3	6.2	92.5	11.7	47.6	79.0	92.5	0.60	15.1
Gemini-2.0-flash	18.2	90.9	30.3	62.9	87.4	90.9	0.72	5.0
Claude-3.5-haiku	15.2	89.5	25.9	60.1	88.1	89.5	0.70	5.9
GPT-4o	10.2	92.3	18.4	49.0	86.7	92.3	0.65	9.0
DeepSeek-R1-Llama3-8b	16.9	74.1	27.5	42.0	73.4	74.1	0.54	**4.5**
Mistral-Large	8.9	90.9	16.2	18.2	84.6	90.9	0.46	10.2
DeepSeek-R1	14.0	95.1	24.5	56.6	93.0	95.1	0.69	6.8
Gemini-2.5-pro	13.3	94.4	23.3	58.0	90.9	94.4	0.71	7.1
Claude-3.7-sonnet	10.2	97.9	18.4	60.8	93.0	97.9	0.73	9.6
GPT-o1	14.3	92.3	24.7	54.5	88.8	92.3	0.67	6.5
GPT-o3	12.1	97.2	21.6	58.7	93.0	97.2	0.72	8.0
SmartAuditFlow (GPT-4o)	18.1	**100**	30.7	**66.4**	**99.3**	**100**	**0.80**	5.5

Note: Placeholders (-) indicate data is not available.

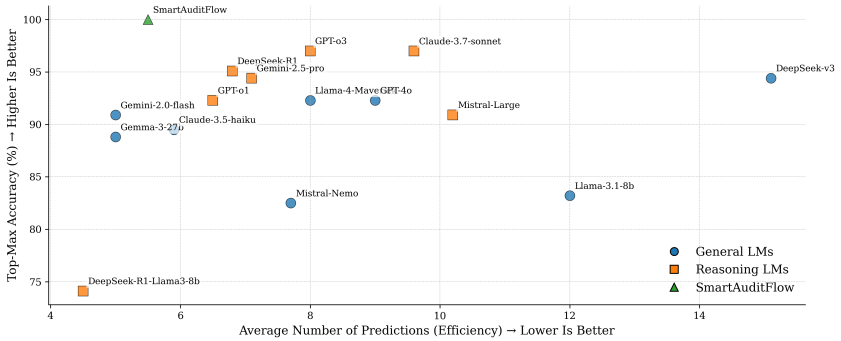


Fig. 4. Trade-off Between Accuracy and Efficiency on Standard Vulnerability Set

accuracy (94-97%) with a more moderate average of 6.5 to 9.6 findings. SmartAuditFlow stands out by achieving the highest top-*max* accuracy (100%) while generating an average of only **5.5** findings per contract.

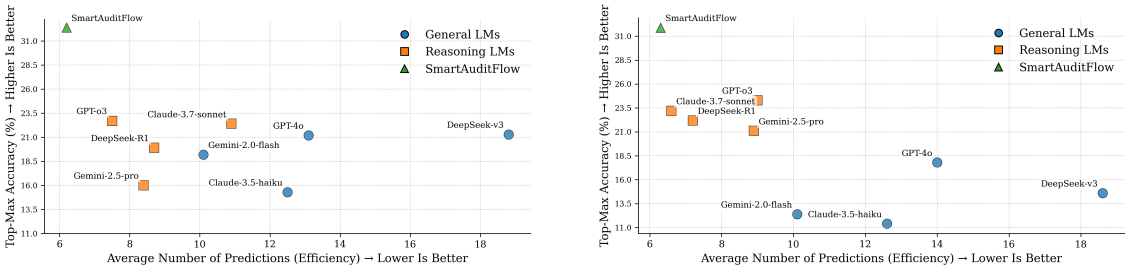
> **Answer to RQ2:** SmartAuditFlow excels in ranking critical vulnerabilities, achieving a top-*max* accuracy of 100%. Its high MRR of 0.8 indicates that the first true positive is typically found very early in the ranked list (average rank 1.25). Furthermore, SmartAuditFlow demonstrates superior efficiency by achieving this high level of detection accuracy while generating a significantly lower average number of findings per contract (5.5) compared to other standalone LLM approaches.

4.4.3 RQ3: Performance on Real-World Projects. While RQ1 and RQ2 demonstrated strong performance on datasets with common, often well-defined vulnerability types, real-world smart contracts frequently present more complex, nuanced, and unique security issues. To address RQ3, we conducted a comprehensive evaluation of SmartAuditFlow and selected baseline methods using using two real-world datasets: Code4rena dataset and Sherlock dataset. This

Table 2. Comparative Performance of Vulnerability Detection Tools on the Code4rena and Sherlock Datasets

Tool	Code4rena					Sherlock				
	P(%)	R(%)	F1(%)	Avg. Outputs	MAP	P(%)	R(%)	F1(%)	Avg. Outputs	MAP
DeepSeek-v3	3.7	21.3	6.3	18.8	0.148	2.2	14.6	3.9	18.6	0.139
Gemini-2.0-flash	6.2	19.2	9.4	10.1	0.203	3.5	12.4	5.5	10.1	0.191
Claude-3.5-haiku	3.9	15.3	6.3	12.5	0.123	2.6	11.4	4.2	12.6	0.133
GPT-4o	5.3	21.2	19.6	13.1	0.170	3.6	17.8	6.0	14.0	0.165
DeepSeek-R1	7.5	19.9	10.8	8.7	0.267	8.8	22.2	12.6	7.2	0.241
Gemini-2.5-pro	6.2	16.0	10.8	8.4	0.199	6.7	21.1	10.2	8.9	0.189
Claude-3.7-sonnet	6.6	22.4	10.2	10.9	0.211	10.0	23.2	14.0	6.6	0.211
GPT-o3	8.3	22.7	10.3	7.5	0.236	7.7	24.3	11.7	9.0	0.223
SmartAuditFlow (GPT-4o)	**17.3**	**32.4**	**22.6**	**6.2**	**0.334**	**14.4**	**31.9**	**19.8**	**6.3**	**0.314**

Note: R(%) equals Accuracy (top-max). Placeholders (-) indicate data is not available.



(a) Code4rena dataset

(b) Sherlock dataset

Fig. 5. Trade-off Between Accuracy (top-max) and Efficiency

evaluation primarily utilizes ranking-based and conventional detection approach to assess overall performance on these challenging contracts.

Based on findings from previous experiments (RQ1, RQ2) and their prevalence, we selected a representative set of standalone LLMs for comparison: General-Purpose models: DeepSeek-v3, Gemini-2.0-flash, Claude-3.5-haiku, GPT-4o; Reasoning models: DeepSeek-R1, Gemini-2.5-pro, Claude-3.7-sonnet, GPT-o3. We also evaluated traditional static analysis tools, Slither and Mythril, on this Real-World Contracts Set. Consistent with findings in related work [10, 30], these tools exhibited very limited capability in detecting the complex logic vulnerabilities, often failing to identify any of the nuanced issues present.

The performance results on both datasets are summarized in Table 2. On Code4rena dataset, SmartAuditFlow achieved a top-*max* accuracy of 32.4% and an MAP of 0.334, while generating the lowest average number of outputs per contract at 6.2. On Sherlock dataset, SmartAuditFlow maintained competitive performance with a top-*max* accuracy of 31.9% and an MAP of 0.314, again producing only 6.3 outputs per contract on average. In both cases, the system outperformed all baselines by a significant margin, with relative improvements in F1-score ranging from 5.8 to 16.4 percentage points over the best-performing standalone LLM.

The MAP results further demonstrate SmartAuditFlow's ranking advantage. On Code4rena, it achieved 0.334, surpassing all baselines (e.g., GPT-o3: 0.236, DeepSeek-R1: 0.267, Claude-3.7-sonnet: 0.211, GPT-4o: 0.170). On Sherlock, it reached 0.314, again outperforming GPT-o3 (0.223), DeepSeek-R1 (0.241), and all general-purpose models (< 0.20). These scores imply that the first correct finding typically appears within the top three outputs, whereas strong baselines often rank it fourth or lower, underscoring SmartAuditFlow's superior prioritization of true positives.

Figure 5 illustrates the trade-off between top-*max* accuracy and prediction efficiency (average outputs per contract) for both datasets. While many standalone LLMs struggled to achieve high accuracy without producing large output volumes, SmartAuditFlow achieved the highest accuracy with minimal findings on both platforms.

Answer to RQ3: On two challenging real-world datasets, SmartAuditFlow demonstrates better performance than standalone LLM applications and traditional tools. It achieves top-*max* detection accuracies of 32.4% (Code4rena) and 31.9% (Sherlock) with MAP scores of 0.334 and 0.314 respectively, while maintaining high efficiency with only about 6 findings per contract. These results confirm that SmartAuditFlow generalizes well to heterogeneous, professionally audited projects, validating its practical applicability for complex, real-world smart contract.

4.4.4 RQ4: Impact of Different LLM Backbones on SmartAuditFlow Performance. SmartAuditFlow is designed as a versatile framework whose performance can be influenced by the choice of the underlying LLM and other operational parameters. To address RQ4, we specifically investigate the impact of varying the LLM backbone on SmartAuditFlow's vulnerability detection accuracy and operational efficiency. For this analysis, all other parameters of the SmartAuditFlow workflow, especially the optimized prompt sets, were held constant to isolate the effect of the LLM choice. The evaluations for RQ4 were conducted on the Code4rena dataset to assess performance on complex, practical scenarios.

We configured SmartAuditFlow with four distinct high-performance LLMs: GPT-4o, DeepSeek-v3, Claude-3.7-sonnet, and Gemini-2.5-pro. Performance was compared across four key metrics: top-*max* accuracy, MAP for ranking quality, average number of findings reported per contract, and average inference steps required by SmartAuditFlow per contract. In addition, to address scalability and cost considerations, we measured the *Cost Range* per contract, the *Average Cost* across the dataset, and the *Average Execution Time* from initial plan generation to final audit report.

The results in Table 3 reveal that SmartAuditFlow's performance, while generally strong, varies meaningfully with the integrated LLM. SmartAuditFlow configured with Gemini-2.5-pro attained the highest top-*max* accuracy at **37.16%**, closely followed by Claude-3.7-sonnet configuration at **36.82%**. However, the Gemini configuration also produced the highest average number of findings per contract (16.9), suggesting that its high recall might come with increased output verbosity, potentially requiring more auditor review time. In contrast, the GPT-4o configuration, while achieving a top-*max* accuracy of 32.4%, excelled in ranking quality, securing the highest MAP of **0.334**.

From a cost-efficiency perspective, GPT-4o offered the lowest average cost among high-performing models at **$0.277** per contract, with a cost range of [$0.088, $0.816] and an average execution time of 238.5 s. In contrast, Claude-3.7-sonnet, while strong in accuracy and MAP, had a higher average cost ($0.991) and cost range extending up to $2.314, reflecting its higher token usage. Gemini-2.5-pro maintained competitive costs ($0.337 average) and the fastest runtime (213.2 s), making it a strong candidate for time-sensitive audits. DeepSeek-v3 exhibited the lowest monetary cost ($0.102) but at the expense of significantly lower detection performance.

These findings indicate that SmartAuditFlow can be tuned for different operational priorities: selecting Gemini-2.5-pro for maximum accuracy with low runtime, GPT-4o for balanced accuracy and ranking quality at a moderate cost, or Claude-3.7-sonnet for a strong balance of accuracy and recall where higher costs are acceptable.

Answer to RQ4: SmartAuditFlow's performance is notably influenced by the choice of its LLM backbone, allowing for optimization based on specific auditing priorities. Gemini-2.5-pro achieves the highest top-*max* accuracy (37.16%) and fastest average runtime (213.2 s) but generates more verbose outputs. GPT-4o provides superior

Table 3. Evaluation of Different LLM Backbones for SmartAuditFlow

LLM Config.	Avg. (top-max)	Avg. Outputs	MAP	Avg. Steps	Cost Range	Avg. Cost	Median Cost	Avg. Exec. Time(s)
GPT-4o	32.4%	**6.2**	**0.334**	36.0	[$0.088, $0.816]	$0.277	$0.257	238.51
DeepSeek-v3	25.1%	9.6	0.174	30.6	[$0.010, $0.359]	**$0.102**	**$0.085**	508.21
Claude-3.7-sonnet	36.8%	12.8	0.315	31.9	[$0.161, $2.314]	$0.991	$0.900	381.13
Gemini-2.5-pro	**37.2%**	16.9	0.253	**29.8**	[$0.158, $2.882]	$0.337	$0.275	**213.24**

ranking of critical vulnerabilities with concise reports and competitive costs. Claude-3.7-sonnet offers a strong balance between high detection accuracy and recall, although at higher computational cost. These results enable informed selection of LLM backbones according to project-specific accuracy, budget, and time constraints.

4.4.5 RQ5: Comparison with Other LLM-based Auditing Methods. To contextualize SmartAuditFlow's capabilities within the evolving landscape of LLM-assisted smart contract auditing, we compare its performance against several notable prior works. Notably, David et al. [9], PropertyGPT [25], and GPTScan [30] have demonstrated promising results using LLMs in different auditing scenarios.

David et al. [9] employed direct prompting of GPT-4 and Claude-v1.3 for a binary classification task on 146 vulnerabilities across 38 types, identifying 58 TPs. PropertyGPT [25] focused on verifying LLM outputs, correctly identifying 9 out of 13 CVEs (focused on 5 classic types) and 17 out of 24 vulnerabilities from their SmartInv dataset. GPTScan [30] targeted 10 common logic vulnerabilities, reporting 40 TPs. Their stated recall of 83.33% suggests these 40 TPs were out of approximately 48 actual instances of those types within their specific evaluation context. In contrast, SmartAuditFlow-Enhanced was evaluated on the 784 high and medium severity vulnerability instances within our challenging Real-World Contracts Set, identifying 310 TPs. A key distinction of our work is the aim to address a broad spectrum of vulnerability types ("All Types"), rather than a predefined subset of common or logic vulnerabilities.

To facilitate a more direct comparison on a common ground, we evaluated SmartAuditFlow-Enhanced on the set of 13 representative CVEs previously analyzed by PropertyGPT [25]. We compare our results with those reported for PropertyGPT, GPTScan, Slither, and Mythril (sourced from PropertyGPT's study), and further benchmark against standalone invocations of reasoning-enhanced LLMs: GPT-o3 and DeepSeek-R1. The detailed detection results are presented in Table 4. SmartAuditFlow-Enhanced correctly detected all **13 out of 13 CVEs** in this benchmark, achieving a 100% detection rate. This includes complex cases that other methods reportedly missed (e.g., CVE-2021-3004, CVE-2018-17111, as noted for PropertyGPT which detected 9/13).

Among other LLM-based approaches on this CVE set, standalone prompting of reasoning-enhanced models like GPT-o3 (using its specified version) and DeepSeek-R1 (version) also demonstrated strong capabilities, reportedly identifying 11/13 TPs and 9/13 TPs, respectively, in a single pass (pass@1).

Answer to RQ6: When compared to other LLM-based auditing methods, SmartAuditFlow-Enhanced demonstrates a broader scope of vulnerability detection and achieved a high detection count (310 TPs) on a large set of real-world vulnerabilities. On a standardized 13-CVE benchmark, SmartAuditFlow-Enhanced achieved a 100% detection rate, surpassing reported results for PropertyGPT (9/13), standalone reasoning LLMs like GPT-o3 (11/13), and traditional tools.

4.4.6 RQ6: Ablation Studies. To address RQ5, we investigated the impact of incorporating external knowledge on the performance of SmartAuditFlow by comparing its accuracy and efficiency with and without external knowledge,

Table 4. Vulnerability Detection Performance on 13-CVE benchmark

CVE	Description	Our work	David	PropertyGPT	GPTScan	Slither	Mythril	GPT-o3	Deepseek-R1
CVE-2021-34273	access control	✓	×	✓	✓	×	×	✓	✓
CVE-2021-33403	overflow	✓	×	✓	×	×	✓	✓	✓
CVE-2018-18425	logic error	✓	×	✓	×	×	×	×	×
CVE-2021-3004	logic error	✓	×	×	×	×	×	×	×
CVE-2018-14085	delegatecall	✓	✓	×	×	✓	×	✓	✓
CVE-2018-14089	logic error	✓	✓	✓	✓	×	×	✓	×
CVE-2018-17111	access control	✓	✓	×	×	×	×	✓	✓
CVE-2018-17987	bad randomness	✓	✓	×	✓	×	×	✓	✓
CVE-2019-15079	access control	✓	×	✓	×	×	×	✓	✓
CVE-2023-26488	logic error	✓	×	✓	×	×	×	×	×
CVE-2021-34272	access control	✓	×	✓	✓	×	×	✓	✓
CVE-2021-34270	overflow	✓	✓	✓	✓	×	✓	✓	✓
CVE-2018-14087	overflow	✓	×	✓	×	×	✓	✓	✓

Note: ✓ indicates a correct detection (TP), whereas × indicates an incorrect detection (FN).

including the combination of static tools (see Section 3.2.3) and RAG techniques (see Section 3.3.2). As established in RQ4 results, the baseline configuration of SmartAuditFlow utilized Gemini-2.5-pro as its LLM backbone. We evaluated configurations with static analysis alone, RAG alone, and the combination of both.

As observed in Table 5, integrating external knowledge sources leads to notable improvements in SmartAuditFlow's detection accuracy. Incorporating static analysis tools alone (SmartAuditFlow + static tool) increased the top-*max* accuracy from 37.2% to **39.9%** (a +2.7 percentage point improvement). This enhancement can be attributed to the static analyzer's ability to flag determinate, pattern-based vulnerabilities and provide structural insights (e.g., identifying critical contract functions or inter-contract relationships) that refine the LLM's initial analysis.

Adding RAG alone (SmartAuditFlow + RAG) also improved accuracy, increasing it from 37.2% to **40.1%** (a +2.9 percentage point improvement). This underscores RAG's effectiveness in augmenting the LLM with external, up-to-date knowledge during the calibration phase, thereby enhancing its vulnerability verification capabilities and reducing potential hallucinations by grounding findings in retrieved evidence. The most significant performance was achieved when both static analysis tools and RAG were integrated (SmartAuditFlow + Static Tool + RAG), yielding a top-*max* accuracy of **41.2%**. This represents a substantial +4.0% improvement over the baseline configuration. This result supports our hypothesis that leveraging multiple, complementary external knowledge sources synergistically enhances SmartAuditFlow's overall detection performance.

The average number of outputs per contract remained remarkably stable and slightly decreased with enhanced knowledge integration (from 16.9 to 15.7). While ranking quality (MAP) showed some variation, the full integration of static tools and RAG notably improved this metric (from 0.253 to 0.281). This demonstrates that the accuracy gains from external knowledge do not lead to excessive output, and can concurrently improve ranking, ensuring auditor efficiency.

Answer to RQ5: The integration of external knowledge sources significantly enhances SmartAuditFlow's performance. Incorporating both static analysis tools and RAG techniques boosts the top-*max* detection accuracy by 4.0% increase (from 37.2% to 41.2%) over the baseline configuration.

5 Related Work and Discussion

5.1 Related Work

Recent studies have demonstrated growing potential for applying LLMs to software vulnerability analysis and smart contract security. Liu et al. [23] examined GPT's performance across six tasks—including bug report summarization,

Table 5. Evaluation of SmartAuditFlow Performance with External Knowledge Integration

Configurations	Accuracy (top-max)	Avg. Outputs	MAP
SmartAuditFlow (Baseline)	37.2%	16.9	0.253
SmartAuditFlow + static tool	39.9% (+2.7%)	16.1	0.242
SmartAuditFlow + RAG	40.1% (+2.9%)	16.7	0.257
SmartAuditFlow + static tool + RAG	41.2% (+4.0%)	15.7	0.281

security bug report identification, and vulnerability severity evaluation. They compared GPT's results with 11 SOTA approaches. Their findings indicate that, when guided by prompt engineering, GPT can outperform many existing techniques. In parallel, Yin et al. [41] established benchmark performance metrics for open-source LLMs across multiple vulnerability analysis dimensions. They emphasized how strategic prompt design and model fine-tuning substantially impact detection accuracy and localization precision.

Researchers have extended these investigations to domain-specific languages (DSLs) likeC/C++, Java, and Solidity. Khare et al. [16] conducted an comprehensive evaluation of 16 advanced LLMs in the context of C/C++ and Java, showing that LLM-based approaches can surpass traditional static analysis and deep learning tools. Li et al. [20] integrated GPT-4 with CodeQL, a static analysis tool, to detect vulnerabilities in Java code, achieving an accuracy improvement of over 28% compared to using CodeQL alone.

Building on these foundations, recent advancements employ auxiliary techniques to enhance LLM capabilities. LLM4Vuln [31] employs techniques such as knowledge retrieval, context supplementation, and advanced prompt schemes to boost vulnerability detection across multiple languages, including Java, C/C++, and Solidity. Their results show that GPT-4, when leveraging these enhancements, significantly outperforms competing models like Mixtral and CodeLlama. Similarly, GPTScan [30] demonstrates that a binary (Yes/No) response format can be effectively used by GPT models to confirm potential vulnerabilities in Solidity smart contracts.

Furthermore, PropertyGPT [25] integrates GPT-4 with symbolic execution at the source code level, enabling formal verification of smart contract properties. Meanwhile, Ma et al. [26] propose a paradigm-shifting architecture that decouples detection and reasoning tasks through two-stage fine-tuning. Simultaneously, they improved both accuracy and model interpretability in smart contract analysis.

5.2 Summary of Findings

Based on the above evaluation results, we have derived the several key findings:

- **Superior Vulnerability Detection through a Structured Workflow:** The core innovation of our methodology lies in decomposing the complex audit process into a Plan-Execute paradigm with distinct analytical stages. This structured workflow, mimicking a human auditor's systematic approach, enables the integrated LLM to perform a more thorough and focused analysis of specific code aspects or vulnerability types within each sub-task. This highlights the power of a well-designed workflow in enhancing detection capabilities.
- **Prompt Optimization and LLM Guidance:** The iterative prompt optimization strategy detailed in our Plan Phase is a foundational element contributing to SmartAuditFlow's effectiveness. By systematically generating and selecting optimized prompts for critical stages like Initial Analysis (A1) and Audit Planning (A2), we ensure the LLM's reasoning is precisely guided towards relevant contract features and potential risk areas. By carefully

crafting the prompt, our approach effectively handles ambiguous or complex code segments, leading to improved detection of subtle vulnerabilities.

- **Effective Integration of External Knowledge Integration:** The integration of external knowledge sources demonstrably enhances SmartAuditFlow's performance. Incorporating static analysis tool outputs provides crucial structural and pattern-based insights during the initial analysis (A1), while RAG enriches the calibration step (A3) with up-to-date, context-specific information. This underscores the value of a multi-source, hybrid approach.
- **Modularity and Flexibility with Plug-and-Play Architecture:** SmartAuditFlow's design exhibits significant modularity and flexibility. The framework's compatibility with various leading LLM backbones (e.g., Gemini, GPT, Claude series) allows users to select a model that best aligns with their specific performance priorities, such as maximizing detection accuracy, optimizing ranking quality (MRR), ensuring report conciseness (average outputs), or managing computational effort (average inference steps).
- **Effectiveness on Complex and Edge-Case Scenarios:** SmartAuditFlow demonstrates robust performance not only on standard benchmarks but also on challenging real-world contracts and specific CVEs, which often represent edge cases or complex interaction-dependent vulnerabilities. Its ability to achieve high detection rates in these scenarios indicates its capacity to handle intricacies that often elude traditional tools and simpler LLM applications.

5.3 Threats of Validity

Our proposed system has the following potential limitations:

- *Prompt Limitations for LLM:* While the iterative prompt optimization framework described in Section 3.2 significantly enhances SmartAuditFlow's reasoning accuracy, it can yield prompts that are highly specialized to the characteristics of a particular LLM family (e.g., GPT-series). This specialization maximizes performance for the target backbone but can reduce effectiveness when switching to an alternative model, particularly one with different tokenization, or output formatting tendencies. In such cases, achieving comparable performance often requires re-running the full optimization process, which introduces additional setup time, computational expense, and data preparation effort. This dependency can reduce the `plug-and-play` flexibility and increase deployment cost in scenarios.
- *Computational Resources and Cost:* The multi-stage, iterative nature of SmartAuditFlow, involving multiple LLM calls per contract (about 30-40 steps in RQ4) and potentially intensive prompt optimization, demands considerable computational resources. Our experiments indicate analysis costs ranging from approximately $0.1 USD for simple contracts to upwards of $1 USD for highly complex ones using commercial APIs. While these costs are substantially lower than traditional human-led audits (often thousands of USD), they represent a consideration for widespread adoption, especially for batch processing of numerous contracts.
- *Dependency on External LLM APIs*: SmartAuditFlow's reliance on commercial LLM APIs (e.g., for OpenAI, Google, Anthropic, DeepSeek models) introduces external dependencies. First, availability risk arises from the fact that these APIs are hosted and managed by third-party providers whose service-level guarantees, maintenance schedules, or policy changes are beyond the control of system deployers. Second, cost volatility remains a long-term concern: commercial API pricing structures may change unpredictably, potentially undermining the economic feasibility of sustained or large-scale deployments. Third, data privacy and compliance risks

arise because transmitting proprietary smart contract code or contextual documents to external endpoints may violate contractual obligations or data protection regulations.

- *Static Analysis Limitations*: SmartAuditFlow primarily performs a static analysis of smart contract code—both through its LLM-driven interpretation and its integration of static analysis tools. This approach is inherently limited in detecting vulnerabilities that only manifest during runtime or depend on complex state interactions specific to a live blockchain environment.

6 Conclusion

This paper introduces SmartAuditFlow, a novel Plan-Execute framework for smart contract security analysis that leverages dynamic audit planning and structured execution. Unlike conventional LLM-based auditing methods that follow fixed, predefined steps, SmartAuditFlow dynamically adapts in real-time, generating and refining audit plans based on the unique characteristics of each contract. Its step-by-step execution process enhances vulnerability detection accuracy while minimizing false positives. The framework integrates iterative prompt optimization and external knowledge sources, such as static analysis tools and Retrieval-Augmented Generation (RAG), ensuring security audits are grounded in both code semantics and real-world security intelligence. Extensive evaluations across multiple benchmarks demonstrate SmartAuditFlow's superior performance, achieving 100% accuracy on common vulnerabilities, high precision in real-world projects, and the successful identification of all 13 tested CVEs.

Potential future enhancements include integrating knowledge graphs into the RAG process, adopting formal verification techniques to improve LLM output accuracy, and optimizing prompt strategies for increased efficiency. Additionally, improving audit decision interpretability will be essential for real-world adoption. We envision SmartAuditFlow as a significant advancement in automated smart contract auditing, offering a scalable, adaptive, and high-precision solution that enhances blockchain security and reliability.

Acknowledgments

This paper is supported by National Key Research and Development Program of China under the grant No.2023YFB2703704, Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing under Grant GJJ-25-002, National Natural Science Foundation of China under Grant No. 62372149, No. U23A20303 and No. 62372173. It is also supported by China Scholarship Council (CSC).

References

[1] Maha Ayub, Tania Saleem, Muhammad Janjua, and Talha Ahmad. 2023. Storage state analysis and extraction of Ethereum blockchain smart contracts. *ACM Transactions on Software Engineering and Methodology* 32, 3 (2023), 1–32.

[2] Razvan Azamfirei, Sapna R Kudchadkar, and James Fackler. 2023. Large language models and the perils of their hallucinations. *Critical Care* 27, 1 (2023), 120.

[3] Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. 2021. On the dangers of stochastic parrots: Can language models be too big?. In *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency*. 610–623.

[4] Maciej Besta, Nils Blach, Ales Kubicek, et al. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 17682–17690.

[5] Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, et al. 2024. Smart Contract and DeFi Security Tools: Do They Meet the Needs of Practitioners?. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[6] Chong Chen, Jianzhong Su, Jiachi Chen, et al. 2023. When chatgpt meets smart contract vulnerability detection: How far are we? *arXiv preprint arXiv:2309.05520* (2023).

[7] Yizheng Chen, Zhoujie Ding, Lamya Alowain, et al. 2023. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. 654–668.

[8] Jiale Cheng, Xiao Liu, Kehan Zheng, et al. 2023. Black-box prompt optimization: Aligning large language models without model training. *arXiv preprint arXiv:2311.04155* (2023).

[9] Isaac David, Liyi Zhou, Kaihua Qin, et al. 2023. Do you still need a manual smart contract audit? *arXiv preprint arXiv:2306.12338* (2023).

[10] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*. 530–541.

[11] Shahul Es, Jithin James, Luis Espinosa Anke, and Steven Schockaert. 2024. Ragas: Automated evaluation of retrieval augmented generation. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*. 150–158.

[12] Jinlan Fu, See-Kiong Ng, Zhengbao Jiang, and Pengfei Liu. 2023. Gptscore: Evaluate as you desire. *arXiv preprint arXiv:2302.04166* (2023).

[13] Gaole He, Gianluca Demartini, and Ujwal Gadiraju. 2025. Plan-then-execute: An empirical study of user trust and team performance when using llm agents as a daily assistant. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–22.

[14] Qing Huang, Dianshu Liao, Zhenchang Xing, et al. 2023. Semantic-enriched code knowledge graph to reveal unknowns in smart contract code reuse. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–37.

[15] Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, et al. 2024. Position: LLMs Can't Plan, But Can Help Planning in LLM-Modulo Frameworks. In *International Conference on Machine Learning*. PMLR, 22895–22907.

[16] Avishree Khare, Saikat Dutta, Ziyang Li, et al. 2023. Understanding the effectiveness of large language models in detecting security vulnerabilities. *arXiv preprint arXiv:2311.16169* (2023).

[17] Patrick Lewis, Ethan Perez, Aleksandra Piktus, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[18] Guohao Li, Hasan Hammoud, Hani Itani, et al. 2023. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems* 36 (2023), 51991–52008.

[19] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. 2014. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 661–670.

[20] Ziyang Li, Saikat Dutta, and Mayur Naik. 2024. Llm-assisted static analysis for detecting security vulnerabilities. *arXiv preprint arXiv:2405.17238* (2024).

[21] Han Liu, Daoyuan Wu, Yuqiang Sun, et al. 2024. Using my functions should follow my checks: understanding and detecting insecure OpenZeppelin code in smart contracts. In *33rd USENIX Security Symposium (USENIX Security 24), USENIX Association*. 3585–3601.

[22] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2023), 21558–21572.

[23] Peiyu Liu, Junming Liu, Lirong Fu, et al. 2024. Exploring {ChatGPT's} capabilities on vulnerability management. In *33rd USENIX Security Symposium (USENIX Security 24)*. 811–828.

[24] Yang Liu, Dan Iter, Yichong Xu, et al. 2023. G-eval: NLG evaluation using gpt-4 with better human alignment. *arXiv preprint arXiv:2303.16634* (2023).

[25] Ye Liu, Yue Xue, Daoyuan Wu, et al. 2025. PropertyGPT: LLM-driven Formal Verification of Smart Contracts through Retrieval-Augmented Property Generation. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society.

[26] Wei Ma, Daoyuan Wu, Yuqiang Sun, et al. 2024. Combining Fine-Tuning and LLM-based Agents for Intuitive Smart Contract Auditing with Justifications. *arXiv preprint arXiv:2403.16073* (2024).

[27] Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, et al. 2024. GSM-Symbolic: Understanding the Limitations of Mathematical Reasoning in Large Language Models. *arXiv preprint arXiv:2410.05229* (2024).

[28] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084* (2019).

[29] Paul Stapor, Leonard Schmiester, Christoph Wierling, et al. 2022. Mini-batch optimization enables training of ODE models on large-scale datasets. *Nature Communications* 13, 1 (2022), 34.

[30] Yuqiang Sun, Daoyuan Wu, Yue Xue, et al. 2024. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[31] Yuqiang Sun, Daoyuan Wu, Yue Xue, et al. 2024. Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning. *arXiv preprint arXiv:2401.16185* (2024).

[32] Palina Tolmach, Yi Li, Shang-Wei Lin, et al. 2021. A survey of smart contract formal specification and verification. *ACM Computing Surveys (CSUR)* 54, 7 (2021), 1–38.

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, et al. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[34] Lei Wang, Chen Ma, Xueyang Feng, et al. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science* 18, 6 (2024), 186345.

[35] Xuezhi Wang and Denny Zhou. 2024. Chain-of-thought reasoning without prompting. *arXiv preprint arXiv:2402.10200* (2024).

[36] Jason Wei, Nguyen Karina, Hyung Won Chung, et al. 2024. Measuring short-form factuality in large language models. *arXiv preprint arXiv:2411.04368* (2024).

[37] Zhiyuan Wei, Jing Sun, Zijian Zhang, et al. 2023. Survey on quality assurance of smart contracts. *Comput. Surveys* (2023).

[38] Yixuan Weng, Minjun Zhu, Fei Xia, et al. 2022. Large language models are better reasoners with self-verification. *arXiv preprint arXiv:2212.09561* (2022).

[39] Shihao Xia, Shuai Shao, Mengting He, et al. 2024. AuditGPT: Auditing Smart Contracts with ChatGPT. *arXiv preprint arXiv:2404.04306* (2024).
[40] Shunyu Yao, Dian Yu, Jeffrey Zhao, et al. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems* 36 (2024).
[41] Xin Yin, Chao Ni, and Shaohua Wang. 2024. Multitask-based evaluation of open-source llm on software vulnerability. *IEEE Transactions on Software Engineering* (2024).
[42] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, et al. 2024. Aflow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762* (2024).
[43] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. 2023. Demystifying exploitable bugs in smart contracts. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 615–627.
[44] Tianyang Zhong, Zhengliang Liu, Yi Pan, et al. 2024. Evaluation of openai o1: Opportunities and challenges of agi. *arXiv preprint arXiv:2409.18486* (2024).
[45] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, et al. 2022. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910* (2022).
[46] Mingchen Zhuge, Wenyi Wang, Louis Kirsch, et al. 2024. GPTSwarm: Language Agents as Optimizable Graphs. In *Proceedings of the 41st International Conference on Machine Learning*. 62743–62767.

A Prompt Optimization for Initial Analysis

Objectives: To identify an optimal instruction ρ^* that, when provided to the LLM (gpt-4o-2024-11-20), maximizes the quality of smart contract analysis focusing on function definitions, state variables, modifiers, and events, according to the objective function in Eq. 1. That instruction will be used in Step A1.

A.1 Problem Setup

1. Dataset Preparation: We curated a high-quality dataset of 1,000 smart contract-analysis pairs, reflecting 130+ typical vulnerability types and 200 scenarios. Aim for publicly available, audited contracts where possible (e.g., from OpenZeppelin, well-known DeFi projects, Etherscan verified sources). The dataset was carefully constructed to ensure comprehensive coverage and alignment with industry audit standards, as shown in Table 6.

For each contract c and "expected result" A in D_{task}. This is the most critical and labor-intensive part. Human smart contract auditors and experienced developers will manually create a detailed analysis A. This analysis must explicitly:

- List all function definitions, including their name, visibility, parameters (with types), return values (with types), and a concise description of their purpose and core logic.
- List all state variables, including their name, type, visibility, and a brief description of their purpose.
- List all modifiers, including their name, parameters (if any), and a description of the conditions they enforce or checks they perform.
- List all events, including their name, parameters, and a description of when and why they are emitted.
- A brief summary of how these components interact or potential questions for specific use of contracts.

Table 6. Prompt Optimization Dataset Statistics

Set	Low Comp.	Medium Comp.	High Comp.	Total Contracts	Vuln. Types	Avg. Vuln.
Training Set	400	240	160	800	130	1.5 per
Validation Set	100	60	40	200	130	1.5 per

2. Complexity Metric (Complexity(ρ)):

$$Complexity(\rho) = \omega_1 \cdot TokenCount(\rho) + \omega_2 \cdot SentenceCount(\rho)$$

where $\omega_1 = 0.7$ and $\omega_2 = 0.3$. $TokenCount(\rho)$ is the number of tokens in the instruction ρ, and $SentenceCount(\rho)$ is the number of sentences in the instruction ρ.

3. Instruction Similarity Metric ($sim(\rho, \rho')$): Use sentence embeddings (from a Sentence-BERT model like all-MiniLM-L6-v2) to represent ρ and ρ', then calculate their cosine similarity.

4. Multi-Criteria Scoring Function $f(\rho, c, A)$:

$$f(\rho, c, A) = w_{\text{exec}} \cdot f_{\text{exec}}(\mathcal{M}(\rho, c), A) + w_{\text{log}} \cdot f_{\text{log}}(\mathcal{M}(\rho, c), A)$$

where w_{exec} is 0.7 and w_{log} is 0.3.

(1) Output Alignment Score $f_{exec}(\mathcal{M}(\rho, c))$: Let $O = \mathcal{M}(\rho, c)$ be the LLM's generated analysis. We will parse both O and the expert analysis A to extract structured information about:

- S_F: Set of identified functions (name, params)
- S_V: Set of identified state variables (name, type)
- S_M: Set of identified modifiers (name)
- S_E: Set of identified events (name, params)
- S_Q: Set of additional questions or situations.

$$f_{\text{exec}}(\mathcal{M}(\rho, c)) = w_{\text{cov}} \cdot f_{\text{coverage}}(\mathcal{M}(\rho, c), A) + w_{\text{det}} \cdot f_{\text{detail}}(\mathcal{M}(\rho, c), A)$$

(2) Component Coverage Coverage ($f_{coverage}$): Ensure the LLM output $\mathcal{M}(\rho, c)$ identifies all relevant structural components (functions, state variables, modifiers, events) present in the smart contract c and doesn't list components that don't exist.

For each component type (e.g., functions, variables), compare the extracted set from $\mathcal{M}(\rho, c)$ ($S_F(O)$) with the set from A ($S_F(A)$). Calculate Recall by comparing the sets extracted from O with those from A.

- $P_F = |S_F(O) \cap S_F(A)|/|S_F(O)|$, how many were correct in the number of the LLM output?
- $R_F = |S_F(O) \cap S_F(A)|/|S_F(A)|$, how many were correct in all correct items?
- $F1_F = 2 \cdot P_F \cdot R_F/(P_F + R_F)$

Finally, $f_{coverage}$ can be a weighted average of the F1-scores for each component type (e.g., functions might be weighted higher than events if deemed more critical for the analysis). $f_{coverage} = w_{\text{F}} \cdot F1_F + w_{\text{V}} \cdot F1_V + w_{\text{M}} \cdot F1_M + w_{\text{E}} \cdot F1_E$.

(3) Detail Accuracy Score ((f_{detail})): For each component correctly identified (i.e., true positives from the $f_{coverage}$ step), assess the accuracy and completeness of its description in $\mathcal{M}(\rho, c)$ compared to the detailed description in A. Inspired by RAGAS [11], use another powerful LLM (the "judge model") to evaluate the quality of the primary LLM's description against the expert analysis A. The judge model would be prompted with the primary LLM's description of a component (e.g., function foo), the expert description of foo from A, and a rubric.

The rubric ask the judge to score aspects like:

- Factual Correctness: (e.g., "Does the LLM accurately state the visibility of function foo as per the expert analysis?") - Scale of 1-5
- Completeness of Detail: (e.g., "Does the LLM mention all parameters of function foo listed in the expert analysis?") - Scale of 1-5 or Yes/No.

- Relevance of Information: (e.g., "Is the LLM's description of foo's purpose relevant and free of unnecessary detail when compared to the expert summary?")

Output Likelihood Score $f_{\log}(\mathcal{M}(\rho, c), A)$: Calculated as per Eq. 5 in Section **??** (average negative log-likelihood per token of the LLM's output).

A.2 Procedure

1. Initial Instructions Set $\mathcal{U}_0$. Set initial population size as $k = 20$. Stochastic sampling for Stage A1:

Meta-Prompt: *You are an expert prompt engineer. Generate 20 diverse candidate instructions for an LLM. The LLM's task is to produce a precise and comprehensive analysis of a given smart contract. The analysis must specifically and accurately detail: 1. All function definitions (including name, visibility, parameters, return values, and purpose). 2. All state variables (including name, type, visibility, and purpose). 3. All modifiers (including name, parameters, and conditions enforced). 4. All events (including name, parameters, and emission context). The instructions should be clear, unambiguous, and encourage thoroughness.*

Mutation of Seed Prompts. Manually crafted seed prompts representing different approaches, for example:

- ρ_{seed1}: "Analyze the provided smart contract. Focus on its function definitions, state variables, modifiers, and events. Be precise."
- ρ_{seed2}: "List all functions, state variables, modifiers, and events in the smart contract. For each function, detail its parameters, return values, and purpose. For state variables, list type and purpose. For modifiers, explain their checks. For events, list parameters and when they are emitted."

Apply paraphrasing, keyword substitution (e.g., "list" vs "detail" vs "enumerate"; "purpose" vs "functionality"), and structural edits (reordering points, changing from imperative to declarative). Initial temperature $\tau = 0.7$ for mutation, and replay buffer ($\mathcal{B}_{\text{replay}}$) as empty.

2. Evolution Loop. Max generations: $T = 10$ (max generations).

Mini-Batch sampling: $n_t = \lceil 0.1 \cdot |\mathcal{D}_{train}| \cdot (1 + t/T) \rceil$. Since $|\mathcal{D}_{train}| = 800$, n_t starts at 80 and goes up to 160.

Stochastic fitness evaluation: Use Eq. 2 with $\epsilon = 0.1$ (weight for replay-based regularization).

Elite selection with momentum: $k_e = 10$ (number of elites). $\alpha = 0.3$ (momentum coefficient).

Offspring generation with guided mutation: Generate $k - k_e = 10$ offspring. Set $\tau_{\text{initial}} = 0.7$, $\beta = 0.1$ for exponential decay of mutation temperature ($\tau_t = \tau_{\text{initial}} \cdot e^{-\beta t}$).Prioritize mutations that show improvement on the current mini-batch $\mathcal{B}_t$ for a subset of mutations.

Population Update: $\mathcal{U}_t = \mathcal{E}_t \cup \mathcal{O}_t$. Update $\mathcal{B}_{\text{replay}}$ with the top k_e instructions from $\mathcal{U}_t$ and their $\bar{f}_t(\rho)$ scores if they are better than what's stored.

Convergence Check:

- Terminate if the moving average fitness $\bar{f}_t(\rho)$ of the top elite does not improve by more than $\delta = 0.005$ for 5 consecutive generations.
- Terminate if population diversity $D(\mathcal{U}_t)$ (average pairwise similarity) falls below, e.g., 0.3 (if instructions become too similar).

3. Final Evaluation. Upon termination, take all instructions from the final population $\mathcal{U}_T$.

Evaluate each $\rho \in \mathcal{U}_T$ using the full objective function (Eq. 1) on the held-out validation set $\mathcal{D}_{\text{val}}$. Use regularization hyperparameter $\lambda = 0.01$ for complexity; It needs tuning, could be another parameter sweep.

The ρ^* that maximizes this score on $\mathcal{D}_{\text{val}}$ is selected as the optimal instruction.

4. Hyperparameter Summary (Initial Values). : Population size (k): 20; Elite size (k_e): 10; Max generations (T): 10; Initial mutation temperature (τ_{initial}): 0.7; Mutation temperature decay rate (β): 0.3; Replay regularization weight (ϵ): 0.1; Momentum coefficient (α): 0.3; Complexity regularization (λ): 0.01; Mini-batch base fraction: 0.1; Fitness improvement threshold (δ): 0.005; Scoring weights: $w_{\text{exec}} = 0.7$, $w_{\log} = 0.3$; (within f_{exec}) $w_{cov} = 0.6$, $w_{det} = 0.4$.

5. Evaluation of the Experiment:

- The primary output is the optimal instruction ρ^*.
- Secondary evaluations could include: Tracking the average and best fitness over generations to observe convergence; Analyzing the characteristics of ρ^* (length, specific keywords, structure); Qualitative human evaluation of analyses generated by ρ^* on a few unseen contracts compared to analyses from seed prompts or baseline prompts.

Expected Outcome: The final optimal instructions ρ^* for A1/A2 are released are released in our Github repositories. Table 8 reports a sample of *initial* seed instructions and their Execution/Log/Combined scores across temperatures. The Stage A1 example in Table 7 then traces a single evolutionary trajectory from a generic ρ_0 to ρ^*.

A.3 Example: Stage A1 Prompt Evolution in Practice

```
function withdrawFunds(uint256 _weiToWithdraw) public {
    require(balances[msg.sender] >= _weiToWithdraw);
    require(_weiToWithdraw <= withdrawalLimit);
    require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
    require(msg.sender.call.value(_weiToWithdraw)()); // external call
    balances[msg.sender] -= _weiToWithdraw;          // state update after call
    lastWithdrawTime[msg.sender] = now;
}
```

Listing 1. Reentrancy on withdrawal function

For Stage A1, the primary objective of this stage is to conduct a complete structural analysis of a smart contract Listing 1, identifying its `functions`, `state variables`, `modifiers`, and `events`. The algorithm runs for $T = 10$ generations with a population of $k = 20$ ($k_e = 10$ elites). Key hyperparameters include an initial mutation temperature $\tau_{\text{initial}} = 0.7$ (decaying exponentially with $\beta = 0.3$), replay weight $\epsilon = 0.1$, momentum $\alpha = 0.3$, and complexity regularization $\lambda = 0.01$. Based on a training pool of $|\mathcal{D}_{\text{train}}| = 800$, the sample size per generation, n_t, increases from 80 to 160. Following Eq. 3, fitness is calculated as a weighted sum, $f(\rho, c, A) = 0.7 f_{\text{exec}} + 0.3 f_{\log}$, where the execution fitness is a sub-composition of coverage and detail metrics, $f_{\text{exec}} = 0.6 f_{\text{coverage}} + 0.4 f_{\text{detail}}$.

Step 1 — Initial set $\mathcal{U}_0$ (stochastic sampling + mutation). Using the meta-prompt in Appendix A.2, we sample 20 diverse A1 instructions and then apply paraphrasing, keyword substitution (*list/detail/enumerate*,

purpose/functionality), and structural edits (reordering, imperative ↔ declarative). This yields seeds that vary in explicitness of required fields and output format constraints.

Step 2 — Evolution loop (mini-batch fitness with momentum & replay, elites, offspring). For t=1...T, each $\rho \in \mathcal{U}_{t-1}$ is scored on mini-batch $\mathcal{B}_t$; we compute the momentum-smoothed fitness $\bar{f}_t(\rho)=\alpha \bar{f}_{t-1}(\rho)+(1-\alpha) f_t(\rho)$ and add replay regularization (weight ϵ). We retain k_e=10 elites and produce 10 offspring via guided mutation (prioritizing mutations that increase f on $\mathcal{B}_t$). We terminate if elite moving-average improvement $< \delta$=0.005 for 5 consecutive generations or if diversity $D(\mathcal{U}_t)$<0.3.

Step 3 — Final evaluation (selecting ρ^*). On termination, we evaluate $\mathcal{U}_T$ on $\mathcal{D}_{\text{val}}$ with the full objective (Eq. 1) and select ρ^* that maximizes it.

A1 mini-case (structural analysis on a contract containing `withdrawFunds`). Note that A1 does *not* prove vulnerabilities; it produces a thorough, machine-checkable inventory that later enables A2/A3 to focus on risk (e.g., functions containing external calls before state updates). The table below shows instruction evolution for A1 only; scores follow Eq. 3 with Execution Score decomposed into coverage/detail and Log Score as mean token log-likelihood.

Observations. ρ_0 under-specifies fields and yields partial inventories (missed return types and emission contexts). Guided mutations that (i) enumerate required fields, (ii) demand tabular outputs, and (iii) enforce schema validation substantially raise coverage/detail (Execution). Adding lightweight annotations for *external calls* and *state writes* within function summaries (still within A1 scope) improves the usefulness of A1 outputs for downstream planning without leaking A2/A3 logic into A1. The net effect is a higher Combined score under Eq. 3, aligning with the intended A1 objective.

Table 7. Stage A1 Instruction Evolutionary Trajectory

Instr.	Instruction excerpt	Execution	Log (Mean)	Combined
ρ_0	"Analyze the smart contract and list its functions, state variables, modifiers, and events." (generic; no schema or completeness guarantees)	90.828	-88.628	**36.991**
ρ_4	"Enumerate *all* functions with {name, visibility, typed params, typed returns, *purpose*}; state vars with {name, type, visibility, *purpose*}; modifiers with {name, params, *enforced condition*}; events with {name, params, *emission context*}. Output in 4 tables." (explicit fields & tabular format)	92.812	-54.917	**48.493**
ρ^*	Present in Appendix A.4 (schema + signals useful to later stages)	96.741	-60.102	**49.688**

Table 8. A Sample of Initial Seed Instructions Scores

Instruction	Temp.	Exec. Score	Log Score	Combine	Instruction	Temp.	Exec. Score	Log Score	Combine
ρ_0	0.1	90.828	-88.628	**36.991**	ρ_{10}	0.7	68.710	-52.154	32.451
ρ_1	0.1	90.234	-103.886	**31.998**	ρ_{11}	0.7	71.546	-52.366	34.372
ρ_2	0.1	90.984	-90.116	**36.654**	ρ_{12}	0.7	75.234	-60.393	34.546
ρ_3	0.1	89.718	-105.129	31.264	ρ_{13}	0.7	70.875	-47.927	35.234
ρ_4	0.4	91.056	-109.270	30.958	ρ_{14}	0.7	77.839	-71.377	33.074
ρ_5	0.4	79.779	-74.885	33.380	ρ_{15}	1.0	70.662	-60.360	31.355
ρ_6	0.4	80.375	-73.798	34.123	ρ_{16}	1.0	69.232	-69.027	27.754
ρ_7	0.4	70.532	-108.916	16.698	ρ_{17}	1.0	72.078	-52.578	34.681
ρ_8	0.4	75.615	-73.416	30.906	ρ_{18}	1.0	69.375	-13877.186	-4114.593
ρ_9	0.4	69.578	-76.807	25.663	ρ_{19}	1.0	71.399	-11339.222	-3351.787

A.4 Optimal Prompt Templates for A1/A2

Action 1. Context-Aware Initial Analysis Prompt

System Instruction: You are an expert in solidity programming language and smart contract analysis.
User Instruction: Given the following smart contract code snippet {context} and static analysis: {static_tool}.
Please perform a thorough audit of the code by following these steps:

1. **Careful Reading:**
   - Read through the entire code snippet meticulously.
   - Pay attention to function definitions, state variables, modifiers, and events.
2. **Summarize Key Components:**
   - Break down the contract into its main components.
   - Identify and list key elements such as:
     - Contract Name
     - Purpose
     - Inheritance and Imports
     - State Variables
     - Functions and Modifiers
3. **Note External Interactions:**
   - Identify any calls to external contracts or dependencies.
   - Understand how external data or calls influence the contract's behavior.
4. **Identify Special Mechanisms:**
   - Look for implementations of design patterns (e.g., Singleton, Factory, Proxy).
   - Recognize any cryptographic functions or complex algorithms.
5. **Formulate Questions:**
   - Note any parts of the code that are unclear or complex.
   - Prepare questions or points that need further clarification during the audit.
6. **Provide Output in JSON Format:**
   - Compile your findings into a JSON string with the following structure:

Example Output:

```
{
  "ReviewingTheCode": "Summary of your initial code review.",
  "IdentifyingKeyComponents": {
    "ContractName": "Name of the contract",
    "Purpose": "Purpose of the contract",
    "InheritanceAndImports": {
      "InheritsFrom": ["List of inherited contracts"],
      "Imports": ["List of imported interfaces and libraries"]
    },
    "StateVariables": [
      {
        "Name": "Variable name",
        "Type": "Data type",
        "Purpose": "Brief description"
      }
      // Add more state variables as needed
    ],
    "FunctionsAndModifiers": [
      {
        "Name": "Function or modifier name",
        "Visibility": "public/private/internal/external",
        "Purpose": "Brief description",
        "Parameters": ["List of parameters"],
        "Returns": ["List of return values"]
```

```
        }
        // Add more functions or modifiers as needed
      ]
    },
    "ExternalInteractions": ["List any external contracts or dependencies"],
    "SpecialMechanisms": ["Describe any special mechanisms or patterns used"],
    "Questions": ["List any questions or unclear aspects"]
  }
```

7. **Summarize Understanding:**
   - Ensure that the JSON output accurately reflects your understanding of the contract's purpose and key components.
   - Be concise but thorough in your descriptions.

Action 2: Adaptive Audit Planning Prompt

System Instruction: You are an expert in solidity programming language and smart contract analysis.
User Instruction:

1. **Review the Code Snippet and Initial Analysis:**
   - **Code Snippet:**
     - Carefully read the provided {{#code_snippet#}}.
     - Pay attention to functions, state variables, modifiers, and any complex logic.
   - **Initial Analysis:**
     - Refer to {{#analysis_text#}}, including key components and any noted concerns.
     - Focus on areas highlighted as potentially problematic or complex.
2. **Identify Key Audit Areas:**
   - Based on the code and initial analysis, determine the most relevant areas to focus on during the audit.
   - Consider aspects such as:
     - **Security Vulnerabilities:** Risks like reentrancy, overflows, unchecked calls.
     - **Compliance:** Adherence to applicable standards (e.g., ERC-20/721).
     - **Complex Logic/Algorithms:** Bugs from custom implementations.
     - **External Interactions:** External contract calls or data usage.
     - **Access Control:** Ownership and role-based access controls.
     - **Gas Optimization:** Identify inefficiencies.
     - **Liquidity Pools & AMM Risks:** Protect against manipulation and impermanent loss.
     - **Flash Loan Protection:** Prevent flash loan-based attacks.
     - **Oracle Integrity:** Ensure tamper-proof price feeds.
     - **Governance & Voting Security:** Review decentralized governance for vulnerabilities.
     - **Staking/Rewards:** Ensure secure reward distribution and pool protection.
     - **Tokenomics Risks:** Avoid inflationary token models.
     - **Transaction Ordering & Front-running:** Prevent front-running through *commit-reveal* schemes.
     - **Exit Scams/Rug Pulls:** Safeguard against fund drainage or malicious exits.
     - **Cross-Chain Security (if applicable):** Ensure safe cross-chain operations.
     - **Fallback/Emergency Stops:** Ensure emergency halt mechanisms.
3. **Create the Task List `{task_list}`:**
   - For each identified area, formulate specific, actionable audit tasks.
   - The tasks should be clear and directly related to the issues identified.
   - Structure the {task_list} as an array of tasks grouped by audit area.

   Output Format:

```
{
  "task_list": [
    {
      "Area": "Name of the audit area (e.g., 'Security Vulnerabilities')",
      "Tasks": [
```

```
            "First specific task to be performed in this area.",
            "Second specific task."
          ]
        }
        // Add additional areas and tasks as needed.
      ]
    }
```

4. **Prioritize and Refine Tasks:**
   - Review the tasks to ensure they are:
     - **Relevant:** Directly address issues from the initial analysis.
     - **Specific:** Clearly define what needs to be audited.
     - **Actionable:** Provide a clear course of action for the auditor.

B Retrievable Domain Knowledge Base

Goal: To construct a comprehensive, accurate, up-to-date, and efficiently retrievable knowledge base focused on smart contract security, vulnerabilities, best practices, and exploits, serving as the foundation for our Retrieval-Augmented Generation (RAG) system.

B.1 Foundation and Planning

The scope and objectives of the knowledge base were clearly defined, and a plan was developed to ensure its success. The primary scope and objectives include:

- Blockchain Platforms: Initially focus on Ethereum, then potentially expand for future.
- Smart Contract Languages: Primarily Solidity, as it is the most prevalent language for smart contracts.
- Vulnerability Categories: Prioritization of common and critical vulnerabilities (e.g., Reentrancy, Integer Over/Underflow, Access Control, Unchecked External Calls, Gas Limit Issues), while aiming for comprehensive coverage of known smart contract weaknesses.
- Knowledge Types: A diverse range of information sources are targeted:
  - Formal vulnerability definitions and classifications (e.g., SWC Registry, relevant CWEs).
  - Official security guidelines and best practices from authoritative bodies (e.g., Ethereum Foundation, Consensys, OpenZeppelin).
  - Seminal and contemporary research papers on smart contract security and analysis.
  - Detailed technical blog posts, articles, and post-mortems dissecting real-world exploits and novel vulnerabilities from reputable security researchers and firms.
  - Repositories of secure coding patterns and anti-patterns.
  - Publicly available smart contract auditing reports and findings.
  - Relevant sections of smart contract language documentation focusing on security considerations.

Curation Criteria. To ensure the quality, relevance, and reliability of the knowledge base, the following curation criteria were established:

- Source Authority: Prioritization of official documentation, peer-reviewed academic papers, reports from reputable security firms (e.g., Trail of Bits, Sigma Prime, Consensys Diligence), and content from well-known, respected security researchers.
- Accuracy and Verifiability: Information must be accurate and, where possible, cross-verified with multiple sources. Claims should be supported by evidence or clear reasoning.
- Recency and Relevance: For exploit descriptions, emerging vulnerabilities, and evolving best practices, newer information is generally preferred. Foundational concepts may come from seminal works. All content must be directly relevant to smart contract security.
- Clarity and Detail: Preference for sources that provide in-depth explanations, illustrative code examples (vulnerable and remediated), and actionable mitigation strategies.

Data Schema. For each entry/document chunk into the knowledge base, the data schema is designed to capture the following key fields:

- content: The actual text chunk.
- source_url: The original URL or identifier.
- source_type: (e.g., "SWC", "Blog", "Research Paper", "Guideline", "CVE").
- title: Original title of the document.
- publication_date: (YYYY-MM-DD).
- last_accessed_date: When it was last retrieved/checked.
- vulnerability_tags: (e.g., "reentrancy", "integer_overflow", "access_control").
- platform_tags: (e.g., "ethereum", "solidity").
- severity_keywords: (e.g., "critical", "high", "medium", "low",informational", "ground").
- chunk_id: Identifier for the specific text chunk.
- document_id: Identifier for the parent document.
- summary: A brief LLM-generated summary of the chunk.

B.2 Content Acquisition and Ingestion

The process of acquiring and ingesting content involves systematic collection from diverse sources, followed by extraction and cleaning.

Source Collection: A curated list of target websites, databases, academic journals, and code repositories was compiled based on the established curation criteria. Examples of these sources are provided in Table 9. Content acquisition is achieved through:

- Automated Methods: Utilizing web scraping tools (Python libraries: BeautifulSoup) and APIs to systematically gather information from predefined sources.
- Manual Curation: Supplementing automated collection by manually gathering specific high-value documents, such as PDFs of research papers or audit reports, especially from sources where scraping is not feasible or permitted.

Data extraction and cleaning: Raw content extracted from various sources undergoes a rigorous cleaning process:

- Text Extraction: Conversion of diverse formats (HTML, PDF, Markdown) into plain text.
- Boilerplate Removal: Elimination of irrelevant content such as website navigation menus, advertisements, footers, and excessive formatting.
- Normalization: Standardizing whitespace, character encodings, and resolving minor formatting inconsistencies to prepare the text for further processing.

The cleaned and structured data is then managed through a data pipeline for ingestion into the knowledge base.

Table 9. Illustrative Examples of Domain Knowledge Sources for Smart Contracts

Category	Source Name	Description/Relevance	Example URL/Identifier
Official Docs	Solidity Language	Security considerations, language specifics	docs.soliditylang.org/en/latest/security-considerations.html
Vulnerability DB	SWC Registry	Smart Contract Weakness Classification	securing.github.io/SCSVS/
Vulnerability DB	DASP-Top-10 (OWASP)	Top 10 vulnerabilities (older but foundational)	www.dasp.co/
Security Guidelines	Consensys	Known attacks, best practices	consensys.io/diligence/blog
Security Guidelines	OpenZeppelin	Secure development guides, contract standards	openzeppelin.com/contracts
Research	Academic Papers	Peer-reviewed security analyses, new techniques	arXiv, IEEE, ACM Digital Library
Technical Blogs	Trail of Bits Blog	In-depth vulnerability analysis, security tools	blog.trailofbits.com
Technical Blogs	Sigma Prime Blog	Ethereum security, client vulnerabilities	blog.sigmaprime.io
Community Standard	SCSVS	Smart Contract Security Verification Standard	securing.github.io/SCSVS/
ecurity Guidelines	2024 Web3 Security Report	Smart Contract Auditing Tools Review	https://hacken.io/discover/audit-tools-review/
Audit Reports	Various Firms	Public reports from reputable auditors	(CertiK, PeckShield, Code4rena, SlowMist,)

B.3 RAG Configuration and Reproduction Steps

Configuration Overview. The RAG pipeline is implemented with a fixed, reproducible configuration: documents are chunked into 512-token windows with 128-token overlap, embedded via `all-mpnet-base-v2` (Sentence-BERT), and indexed with FAISS HNSW (`M`=64, `efConstruction`=128). At query time, the system retrieves the top-k = 5 most similar chunks (cosine similarity, threshold 0.72), applies optional MMR re-ranking ($\lambda = 0.7$), and concatenates up to 1,800 tokens of evidence in rank order. This configuration balances retrieval accuracy, diversity, and token efficiency; see Table 10 for the complete parameter set.

Once cleaned, the textual data is processed to optimize it for retrieval by the RAG system.

Document Parsing. The cleaned text is parsed to identify its internal structure, such as headings, paragraphs, lists, and code blocks. Special attention is given to code snippets to preserve their formatting and potentially identify the programming language.

Content Chunking. We adopt a *sliding-window* chunking approach as the default for RAG, using 512-token windows with a 128-token overlap, measured with the embedding model's tokenizer to ensure consistency. This overlap preserves context across chunk boundaries without excessive redundancy. Where documents have strong internal topical boundaries (e.g., sections with clear headings), we apply *semantic chunking* using LangChain's text-splitter modules to group content into coherent units before applying the sliding window. If semantic chunking yields segments exceeding 512 tokens, we further subdivide them while retaining a 128-token overlap.

Table 10. RAG configuration used for calibration.

Component	Setting
Chunking	Sliding window, 512 tokens, 128 overlap
Embedding model	`sentence-transformers/all-mpnet-base-v2`
Embedding dim	768
Index	FAISS HNSW (`M=64`, `efConstruction=128`)
Similarity	Cosine (normalized vectors)
Retrieval k	5 (score threshold 0.72, fallback 0.68)
Re-ranking	MMR, $\lambda = 0.7$ (optional)
Max retrieved context	1,800 tokens (concat in rank order)
Update cadence	Rebuild if > 2% KB changes
Knowledge sources	SWC Registry, CWE/CVE notes, EF/Consensys guides, vetted blogs

Embedding Generation. For each processed text chunk, a dense vector embedding is generated using a pre-trained sentence transformer model (embedding models suitable for semantic similarity tasks). These embeddings capture the semantic meaning of the text chunks.

Storage and Indexing. The text chunks, along with their corresponding vector embeddings and associated metadata (as defined in the data schema), are ingested and stored in a specialized Vector Database (Milvus). The Vector Database is configured with an appropriate indexing strategy (HNSW) to enable efficient and scalable approximate nearest neighbor (ANN) search. This allows for rapid retrieval of the most semantically similar document chunks in response to a query. Hybrid search capabilities, combining semantic similarity with keyword-based filtering on metadata, are also leveraged where available.

Retrieval Parameters. At query time, we retrieve the top-$k = 5$ most similar chunks by cosine similarity, applying a minimum similarity threshold of 0.72. If fewer than 3 chunks pass this threshold, we relax it to 0.68. An optional Maximal Marginal Relevance (MMR) re-ranking step ($\lambda = 0.7$) reduces redundancy when many high-similarity chunks are near duplicates. The retrieved snippets (max 1,800 tokens concatenated in similarity order) are prepended to the calibration prompt in Stage A3, with each snippet wrapped in a source tag: `[SRC: <id> (<url>), Sect <section>, Tokens <start:end>]`. All data is saved in pinecone.io.

C Case Study

C.1 Mini Case Study on Reentrancy

Context. Reentrancy remains one of the most impactful classes of smart contract vulnerabilities. We illustrate how *SmartAuditFlow* detects and mitigates a canonical instance using a compact EtherStore-style withdrawal routine, as list in Code 1. The external call forwards control to `msg.sender` before internal state is decremented, enabling an attacker to re-enter `withdrawFunds`, drain funds, and only then execute state updates.

A1-1. Pre-Audit Triage (Static Feature Extraction). We first run Slither to build a structural summary:

- *Callgraph/CFG features:* identify external call sites (`call.value`, `call{}`, `delegatecall`), state writes after external calls, and potential write-after-call patterns.
- *Interface and role cues:* fingerprint common standards and roles via function signatures and modifiers:

(1) **Token standard indicators:** ERC-20 (`totalSupply`, `transfer`, `approve`, `transferFrom`), ERC-721/1155 selectors, `permit` for EIP-2612.
(2) **Governance/privilege:** `Ownable`, `AccessControl`, `Pausable`, custom admin modifiers.
(3) **Upgradeability:** UUPS/Transparent Proxy patterns (`proxiableUUID`, `upgradeTo`, storage gap).
(4) **DeFi interactions:** router-/pool-like calls (e.g., `swapExactTokensForTokens`, LP mint/burn), oracle reads, lending ops.

- *Asset surface:* payable functions, `address(this).balance`, token holdings, functions that transfer ETH/tokens, and state that gates withdrawals (e.g., `withdrawalLimit`, `lastWithdrawTime`).
- *Risk cues for reentrancy:* unguarded external calls before effects, `send`/`call` usage, missing `nonReentrant`, and CEI violations.

A1-2. Business-Logic and Role Reasoning (LLM reasoning over Static Signals). The LLM combines both the Slither summary (symbol table, modifiers map, external call sites, interface matches) and LLM's domain knowledge to infer:

- *Contract type & roles:* not a token/governance/upgradeable proxy; acts as a time-locked ETH savings/escrow with depositor/withdrawer role symmetry.
- *Key functionalities:* deposit accounting via `balances`, throttled withdrawals via `withdrawalLimit` and per-address `lastWithdrawTime`.
- *Exposed assets and trust boundaries:* on-contract ETH and any tokens incidentally held; privileged state is minimal (no owner-only drains), but `withdrawFunds` is an externally callable money-flow function with an external call to `msg.sender`.
- *Candidate invariants:* (i) CEI must hold for withdrawal; (ii) $\sum$ user balances should never exceed `address(this).balance`; (iii) per-address lock should prevent burst drains without recursion.

A2. Plan Generation (Targeted Checks). The planner materializes concrete sub-tasks:)

(1) Verify CEI ordering in `withdrawFunds` (state updates before interactions).
(2) Check for reentrancy guards or check-effects emulation (e.g., `nonReentrant`, mutex).
(3) Confirm whether the external call is strictly necessary and whether a pull pattern is feasible.
(4) Trace money flow: compute writes to `balances[msg.sender]` and lock updates relative to the external call.
(5) Attempt PoC synthesis: minimal attacker harness with recursive fallback to validate exploitability.

A3. Execute & Calibrate (Joint Reasoning). Given the sub-tasks from A2 (*CEI order*, *guard presence*, *external-call necessity*, *money-flow trace*, *PoC synthesis*), *SmartAuditFlow* runs a two-phase loop per sub-task t_i:

(1) **Assessment (LLM)** on the relevant code slice C_i: the LLM performs a focused security review and emits a *preliminary finding* $e_1^i = \langle \text{issue, severity, rationale}, p_0^i, \mathcal{E}_0^i \rangle$ where $p_0^i \in [0, 1]$ is a confidence score and $\mathcal{E}_0^i$ is evidence (code anchors, dataflows, retrieved citations).
(2) **Calibration (Validators + LLM re-check).** The system re-evaluates e_1^i with rule-based and tool-based signals, yielding e_2^i with updated confidence p_1^i:

- *Rule checks*: CEI ordering, guard requirements (`nonReentrant`/mutex), return-value handling, allowance of push vs. pull payment.
- *(Optional) RAG citations*: confirm pattern matches and recommended remediations from current guidelines.

The LLM then performs a brief *self-consistency pass* against validator outputs to resolve contradictions and tighten the rationale.

Only *etained findings* e_3^i that satisfy the acceptance gates are used to synthesize the calibrated finding v^i.

Example on `withdrawFunds`:

- T_{CEI}: CEI order. *Assessment:* The LLM flags "effects after interaction" (external `call{}` precedes balance decrement), $p_0 = 0.78$, cites CEI guidance (via RAG). *Calibration:* Slither confirms an external call followed by state writes (`balances[msg.sender]-=...` and `lastWithdrawTime=...`); rule-check CEI-Order=fail. Self-consistency raises $p_1 \to 0.96$. *Decision:* v_3 accepted (**Critical**, reentrancy-prone).
- T_{Guard}: guard presence. *Assessment:* LLM notes no `nonReentrant`/mutex, $p_0 = 0.72$. *Calibration:* Slither finds no guard modifiers; rule-check Guard-Check=fail. $p_1 \to 0.91$. *Decision:* v_3 accepted (guard missing).
- T_{Flow}: money-flow trace. *Assessment:* LLM tags `msg.sender` as adversarial sink; updates occur post-call, $p_0 = 0.70$. *Calibration:* Graph confirms write-after-call; optional invariant probe shows potential multi-withdraw before lock update. $p_1 \to 0.90$. *Decision:* v_3 accepted (unsafe flow).
- T_{PoC}: exploit attempt. *Assessment:* LLM proposes minimal attacker with recursive `receive()`. *Calibration:* If harness succeeds on the original and fails on the patch, mark Exploit-Confirmed. Otherwise, retain as Likely with strong static evidence.

A4. Synthesis & Evidence. We consolidate all retained findings $S_3 = \{v^i\}$ from A4 into canonical issues $\mathcal{F} = \{F_k\}$, assign risk/severity, attach a reproducible evidence bundle, and emit a remediation/verification plan.

A4.1 Aggregation & De-duplication. Each v^i is mapped to a canonical issue key $\kappa(v^i)$ based on vulnerability pattern (e.g., SWC tag), callsite, function, and state assets.

A4.2 Risk Scoring & Severity. For each accepted F_k, we compute a risk score that accounts for exploitability, impact, exposure surface, and calibrated confidence.

A4.3 Evidence Bundle (per issue). For traceability and auditability, each F_k ships with a standardized "evidence card" that ties the technical finding to the contract's business logic, roles, and assets identified earlier:

A4.4 Example Evidence Card (`withdrawFunds` reentrancy). **Auto-extracted asset/role map (for this case).**

Dimension	Inferred Facts
Roles/Standards	No ERC-20/721/1155 interfaces; no governance/upgradeability markers; public depositor/withdrawer.
Key Functions	`depositFunds()`, `withdrawFunds()` (payable out), per-address time lock.
Exposed Assets	ETH held by the contract; per-user balances; time-lock state; no privileged drain observed.
Risk Hotspots	External call to `msg.sender` inside `withdrawFunds` before effects; missing `nonReentrant`; CEI violated.

A5. Reports Generation. By fusing static structure with LLM reasoning, calibrating with validators and optional RAG, then aggregating into evidence-rich issues with clear patches and verification, *SmartAuditFlow* produces an auditable, action-ready report with reduced false positives.

C.2 Example Comparison of Reports

Table 11 presents a compact, side-by-side comparison between the verified ground truth audit entries and the corresponding SmartAuditFlow-generated vulnerability reports, together with the alignment assessment. The Ground Truth column summarizes the validated vulnerability descriptions and locations derived from the official dataset annotations or audit results. The Generated Report column provides the simplified output automatically produced by SmartAuditFlow for the same contract. The Comparison Result column indicates the degree of semantic and locational alignment between the two, using concise labels (e.g., "Exact Match", "Partial Match", "Mismatch") based on the evaluation rubric described in Section 4.3.

Table 11. Compact comparison of Ground Truth vs. SmartAuditFlow-generated reports.

ID	Ground Truth	Generated Report	Comparison Result
Beebots.sol	[H-01] Signature malleability of EVM's ecrecover in 'verify()' EVM's 'ecrecover' is susceptible to signature malleability, which allows replay attacks, but that is mitigated here … **[H-02] Arbitrary Transfer of Unowned NFTs** … [H-04] function 'tokenByIndex' treats last index as invalid …	{"Findings":[{"Issue": "Reentrancy Vulnerability in Functions", "Severity": "High","Description": "Functions such as 'withdraw' and 'acceptTrade' are susceptible to reentrancy attacks as …", "Impact": "An attacker could repeatedly call the functions to drain funds from the contract …", "Location": "…", "Relevance": true,"Validation": true, "Note": "…"}, …]}	**Analysis of Alignment Between Issue Report and Response List** — **Issue Analysis and Validation** — **[H-01] Signature malleability of EVM's ecrecover in 'verify()'** … **[H-02] Arbitrary Transfer of Unowned NFTs** **Report Description**: … **Correct Matches**: - **[M-00] Legacy Function Usage**, **Partially Matches**: - *None*, **Missed Issues**: …

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009