# No More Hidden Pitfalls? Exposing Smart Contract Bad Practices with LLM-Powered Hybrid Analysis

XIAOQI LI, Hainan University, China
ZONGWEI LI*, Hainan University, China
WENKAI LI, Hainan University, China
YUQING ZHANG, University of Chinese Academy of Sciences, China
XIN WANG, Hainan University, China

As the Ethereum platform continues to mature and gain widespread usage, it is crucial to maintain high standards of smart contract writing practices. While bad practices in smart contracts may not directly lead to security issues, they elevate the risk of encountering problems. Therefore, to understand and avoid these bad practices, this paper introduces the first systematic study of bad practices in smart contracts, delving into over 47 specific issues. Specifically, we propose SCALM, an LLM-powered framework featuring two methodological innovations: (1) A hybrid architecture that combines context-aware function-level slicing with knowledge-enhanced semantic reasoning via extensible vectorized pattern matching. (2) A multi-layer reasoning verification system connects low-level code patterns with high-level security principles through syntax, design patterns, and architecture analysis. Our extensive experiments using multiple LLMs and datasets have shown that SCALM outperforms existing tools in detecting bad practices in smart contracts.

CCS Concepts: • **Software and its engineering** → **Software verification and validation**; • **Security and privacy** → *Software security engineering*; • **Computing methodologies** → *Natural language processing*.

Additional Key Words and Phrases: Smart Contract, Bad Practice, Large Language Model

**ACM Reference Format:**
Xiaoqi Li, Zongwei Li, Wenkai Li, Yuqing Zhang, and Xin Wang. 2025. No More Hidden Pitfalls? Exposing Smart Contract Bad Practices with LLM-Powered Hybrid Analysis. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX).* ACM, New York, NY, USA, 21 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

With the widespread use of blockchain technology, smart contracts have become an important part of the blockchain ecosystem [38, 55]. Smart contracts are computer programs that automatically execute contract terms, controlling assets and operations on the chain. However, due to their public and immutable code, smart contracts have become a significant target for attackers [3]. A total of 464 security incidents occurred in 2023, resulting in losses of up to $2.486

*Corresponding author

Authors' Contact Information: Xiaoqi Li, csxqli@ieee.org, Hainan University, Haikou, Hainan Province, China; Zongwei Li, lizw1017@hainanu.edu.cn, Hainan University, Haikou, Hainan Province, China; Wenkai Li, cswkli@hainanu.edu.cn, Hainan University, Haikou, Hainan Province, China; Yuqing Zhang, zhangyq@nipc.org.cn, University of Chinese Academy of Sciences, Beijing, China; Xin Wang, 24210839000020@hainanu.edu.cn, Hainan University, Haikou, Hainan Province, China.

billion [61]. The most significant attack occurred on September 23rd when Mixin Network's cloud service provider database was attacked, involving approximately $200 million.

**Bad practices** refer to poor coding habits or design decisions in the development of smart contracts [27, 36]. We categorize bad practices into two types: (1) *Security-related bad practices*, which include actual vulnerabilities that may lead to security incidents (e.g., reentrancy); and (2) *Quality-related bad practices*, which encompass code quality issues that affect maintainability, efficiency, and design quality without directly resulting in exploitable vulnerabilities (e.g., code duplication). While security-related bad practices could potentially lead to future security threats, both types have significant impacts: they can cause performance problems, increase security risks, lead to unpredictable code behavior [22], and create hidden economic dangers due to the disruption of regular smart contract activities [39].

Currently, the security audit of smart contracts mainly relies on manual code review and automated tools [15]. However, these methods have their limitations [54, 60]. Manual code review is inefficient and prone to overlook subtle security vulnerabilities. Existing automated tools primarily rely on pattern matching, which cannot accurately detect complex security issues. Moreover, the types of vulnerabilities that these tools can detect are usually relatively limited and may not be able to identify all potential security problems in smart contracts [6]. To achieve a comprehensive audit, multiple tools may be required, each covering different aspects of security. Therefore, effectively detecting and preventing security issues in smart contracts remains an important issue that needs to be solved.

To address these challenges, we propose SCALM (**S**mart **C**ontract **A**udit **L**anguage **M**odel), a framework with two innovations for smart contract auditing. **First**, a static analysis module constructs an extensible knowledge base by extracting code patterns, converting them into semantic vectors via an embedding model, and dynamically updating the repository with newly identified bad practices through automated SWC-ID annotation. **Second**, a multi-layer reasoning verification system combines Retrieval-Augmented Generation (RAG) with Step-Back prompting, enabling hierarchical reasoning from syntax checks to architectural risk analysis. This framework detects both explicit vulnerabilities and latent design flaws, generating structured audit reports. This approach generates detailed audit reports that document identified issues, assess risk scores, and provide concrete remediation suggestions, achieving improved detection accuracy compared to existing tools while maintaining practical applicability for developers. Our contributions are as follows:

- To the best of our knowledge, we provide the first systematic study of bad practices in smart contracts and conduct an in-depth discussion and analysis on 35 security-related and 12 quality-related bad practices.
- We propose SCALM, an LLM-based framework for smart contract bad practices auditing. This framework integrates context-aware function-level slicing and multi-layer reasoning verification (syntax, design patterns, architecture) to generate structured audit reports.
- We conduct comprehensive experiments across multiple datasets and LLMs. Results demonstrate that SCALM performs well and outperforms existing tools for smart contract bad practice detection. At the same time, ablation experiments reveal that RAG and multi-layer reasoning verification can improve SCALM performance.
- We open source SCALM's codes and experimental data at https://doi.org/10.6084/m9.figshare.28008167.

## 2 Background

### 2.1 Large Language Models

LLMs are trained using deep learning techniques to understand and generate human language. They are typically based on the Transformer architecture, such as ChatGPT [21], BERT [8], and GLM [11]. The training process usually involves

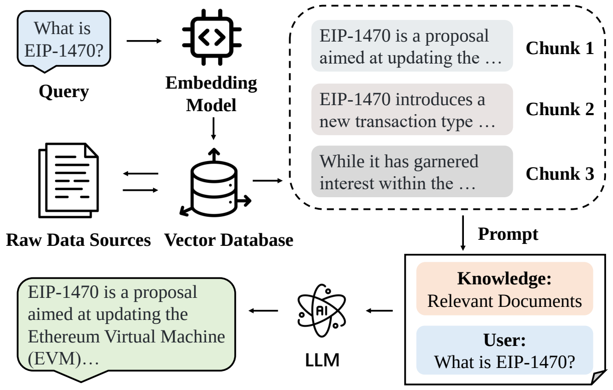

What is EIP-1470?
Query
Embedding Model
Raw Data Sources
Vector Database
EIP-1470 is a proposal aimed at updating the …
Chunk 1
EIP-1470 introduces a new transaction type …
Chunk 2
While it has garnered interest within the …
Chunk 3
Prompt
Knowledge: Relevant Documents
User: What is EIP-1470?
LLM
EIP-1470 is a proposal aimed at updating the Ethereum Virtual Machine (EVM)…

Fig. 1. Illustration of the RAG framework. The system processes user queries through an embedding model, retrieves relevant document chunks from a vector database, and combines the retrieved knowledge with the original query to generate contextually-enhanced responses via LLM.

learning language patterns and structures from large-scale text corpora. These corpora can include a variety of texts such as news articles, books, web pages, and other forms of human linguistic expression. These model can generate coherent and meaningful text by learning from these corpora. Furthermore, LLMs can handle various natural language processing tasks, including text generation, text classification, sentiment analysis, question-answering systems, etc [13].

One of the key features of LLMs is their powerful generative ability. These models can generate new, coherent text similar in grammar and semantics to the training data [49]. This makes LLMs useful for various applications, including machine translation, text summarization, sentiment analysis, dialogue systems, and other natural language processing tasks [51]. Another important feature of LLMs is their "zero-shot" capability, which allows them to perform various tasks without any task-specific training [1]. For example, the model can choose the most appropriate answer given a question and some answer options. This ability makes LLMs very useful in many practical applications.

However, LLMs also have some challenges and limitations. For instance, they may generate inaccurate or misleading information and reflect biases in the training data [53]. LLM can adopt the RAG to improve quality and accuracy. This method combines pre-trained parametric models with non-parametric memory to enhance the quality and accuracy of smart contract code audits [16]. The RAG integrates the processes of retrieval and generation into one.

As Fig. 1 illustrates, during the operation of the model, it first retrieves relevant documents or entities from a large-scale knowledge base. Then, it inputs this retrieved information as additional context into the generation model, which generates corresponding outputs based on these inputs. This design allows RAG to utilize external knowledge bases effectively while demonstrating excellent performance when dealing with tasks requiring extensive background knowledge [10].

## 2.2 Smart Contract Weakness Classification

Smart contracts are self-executing protocols that run on the blockchain and allow trusted transactions without third-party intervention [25, 26]. However, since their code is publicly available and cannot be changed once deployed, the security of smart contracts has become an important issue [59]. To address this issue, EIP-1470 [46] proposes the Smart Contract Weakness Classification (SWC), a classification scheme designed to help developers identify and prevent smart contract weaknesses.

```
contract Proxy {
  address owner;
  constructor() public {
    owner = msg.sender;
  }
  function forward(address callee, bytes _data) public {
    require(callee.delegatecall(_data)); // Allows ANY caller to execute delegatecall to ANY contract
  }
}

contract Proxy_fixed {
  address callee;
  address owner;
  modifier onlyOwner {
    require(msg.sender == owner);
    _;
  }
  constructor() public {
    callee = address(0x0);
    owner = msg.sender;
  }
  function setCallee(address newCallee) public onlyOwner {
    callee = newCallee; // Owner-controlled target update
  }
  function forward(bytes _data) public {
    require(callee.delegatecall(_data)); // Uses preconfigured callee address
  }
}
```

Fig. 2. SWC-112: Delegatecall to Untrusted Callee vulnerability example. The vulnerable Proxy contract allows arbitrary delegatecall execution, while Proxy_fixed implements access control and trusted callee validation to mitigate the security risk.

SWC concerns weaknesses that can be identified within a smart contract's Solidity code [41]. It is designed to reference the structure and terminology of the Common Weakness Enumeration (CWE) but adds several weakness classifications specific to smart contracts [7]. These classifications include but are not limited to, reentry attacks, arithmetic overflow, and delegatecall to untrusted callee. The example in Fig. 2 illustrates how SWC guides secure design: the vulnerable `Proxy` contract allows any caller to invoke `delegatecall` with arbitrary addresses, while the fixed version restricts the `callee` address to an owner-controlled parameter, mitigating unauthorized code execution risks.

All work on SWC has been incorporated into the EEA EthTrust Security Level Specification, a specification proposed by the Enterprise Ethereum Alliance (EEA) to provide a reliable methodology for assessing the security of smart contracts.

## 3 Method

Fig. 3 shows the architecture of SCALM, which establishes a systematic framework for detecting bad practices in smart contracts through multi-layer reasoning and verification. SCALM consists of two core modules : (1) To statically analyze and extract bad practice patterns, convert them into semantic vectors, and construct an extensible knowledge base. (2) Combines RAG and Step-Back prompting in a multi-layer reasoning verification system, which realizes multi-stage verification through layered abstraction from code syntax to architecture. Ultimately, SCALM leverages LLMs to generate structured audit reports.

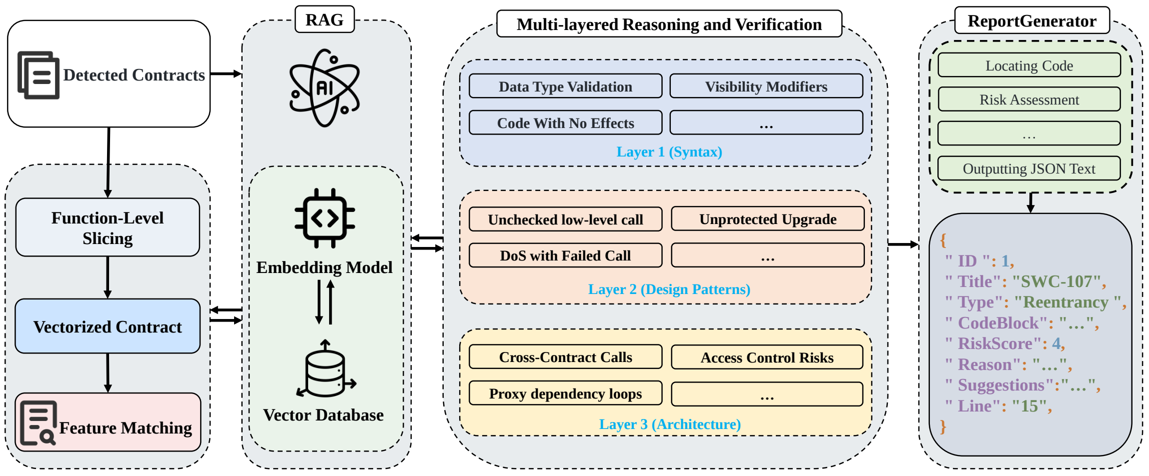


Fig. 3. Overall Architecture of SCALM. The framework comprises context-aware function-level slicing, contract vectorization, RAG-based retrieval, multi-layered reasoning verification (syntax, design patterns, and architecture), and automated report generation with structured JSON output.

## 3.1 Context-Aware Function-Level Slicing

We propose a hierarchical function-level slicing approach that preserves semantic context beyond isolated code units. Unlike traditional program slicing that focuses solely on data/control flow dependencies, our method constructs *context-enriched function slices* that capture both intra-function logic and inter-function call semantics.

Our pipeline illustrated in Fig. 4, operates through three principal stages:

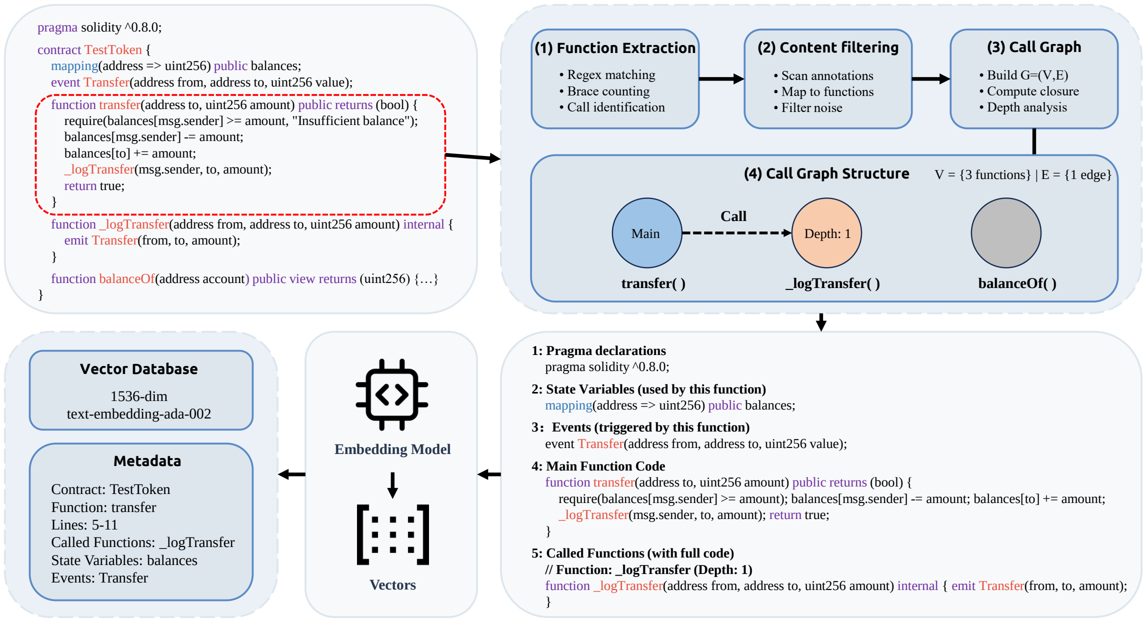
**Function-level Decomposition with Annotation Filtering.** We first decompose the smart contract into independent functions. This is achieved through source code parsing, employing regular expression matching and brace counting techniques to identify function boundaries while preserving relevant documentation precisely. Subsequently, we employ selective extraction for knowledge base construction, focusing on bad practice-related patterns. This step is guided by SWC annotations embedded in the source code (e.g., // SWC-107: L15-20). By mapping the line numbers of these annotations to function boundaries, we construct a filtered corpus containing only functions documented with bad practices, thereby significantly improving the signal-to-noise ratio.

**Dependency-Aware Context Assembly.** For each target function $f_{\text{main}}$, we construct an enriched context $C(f_{\text{main}})$ that captures its semantic environment. This process begins with building a contract-wide directed call graph $G = (V, E)$ to analyze inter-function dependencies. The enriched context is then assembled as follows:

$$C(f_{\text{main}}) = \{\text{pragmas}\} \cup V_{\text{rel}} \cup E_{\text{rel}} \cup \{f_{\text{main}}\} \cup D(f_{\text{main}}, d_{\max}) \tag{1}$$

where $V_{\text{rel}}$ represents state variables referenced within $f_{\text{main}}$, $E_{\text{rel}}$ denotes triggered event definitions, and $D(f, d)$ recursively includes called functions up to a depth of $d_{\max} = 3$. This bounded recursion is formalized as:

$$D(f,d) = \begin{cases} \emptyset & \text{if } d = 0 \\ \bigcup_{g \in \text{calls}(f)} \{g\} \cup D(g, d-1) & \text{otherwise} \end{cases} \tag{2}$$


```
pragma solidity ^0.8.0;

contract TestToken {
    mapping(address => uint256) public balances;
    event Transfer(address from, address to, uint256 value);
    function transfer(address to, uint256 amount) public returns (bool) {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        balances[msg.sender] -= amount;
        balances[to] += amount;
        _logTransfer(msg.sender, to, amount);
        return true;
    }
    function _logTransfer(address from, address to, uint256 amount) internal {
        emit Transfer(from, to, amount);
    }
    function balanceOf(address account) public view returns (uint256) {…}
}
```

```
1: Pragma declarations
   pragma solidity ^0.8.0;
2: State Variables (used by this function)
   mapping(address => uint256) public balances;
3:  Events (triggered by this function)
   event Transfer(address from, address to, uint256 value);
4: Main Function Code
   function transfer(address to, uint256 amount) public returns (bool) {
      require(balances[msg.sender] >= amount); balances[msg.sender] -= amount; balances[to] += amount;
      _logTransfer(msg.sender, to, amount); return true;
   }
5: Called Functions (with full code)
   // Function: _logTransfer (Depth: 1)
   function _logTransfer(address from, address to, uint256 amount) internal { emit Transfer(from, to, amount);
   }
```

Metadata
Contract: TestToken
Function: transfer
Lines: 5-11
Called Functions: _logTransfer
State Variables: balances
Events: Transfer

Fig. 4. Function-Level Context Slicing and Vectorization Pipeline. Illustration of our workflow using the `TestToken` contract as an example. The process begins by extracting a target function (`transfer`) and analyzing its dependencies (e.g., its call to `_logTransfer`). A context-enriched slice is then assembled, incorporating the main function, its dependencies, and relevant contract-level definitions (state variables, events). Finally, this slice is vectorized and stored in a database with structured metadata for retrieval.

Fig. 4 illustrates this process with the `TestToken` contract, the assembled slice is a multi-layered composition including: (1) pragma declarations, (2) relevant state variables (e.g., `balances`), (3) relevant event definitions (e.g., `Transfer`), (4) the main function's code, and (5) the full code of called functions (e.g., `_logTransfer()` at depth 1). The depth limit balances capturing sufficient context while avoiding noise from deep call chains that could reduce embedding quality.

**Structured Metadata Augmentation.** Each context slice is annotated with structured metadata, including source information (source file, contract name), bad practice classification (extracted SWC types), and dependency details (e.g., called functions, referenced state variables, and triggered events). This design, which combines vectorised code content with its corresponding structured metadata, forms a dual representation enabling semantic similarity retrieval and fine-grained filtering during multi-layered reasoning and verification.

## 3.2 Semantic Vectorization and Retrieval

We use a vector database to store and query large amounts of vector data. A vector database is a particular database that can store and query large amounts of vector data [18]. In the vector database, data is stored as vectors, each typically represented by a set of floating-point numbers. These vectors can represent various data types, such as images, audio, text, etc. In bad practice detection tasks, through the embedding model (i.e., `text-embedding-ada-002`), code slices are transformed into 1536-dimensional vectors that preserve syntactic structures and semantic relationships. This process captures both explicit bad practice patterns and implicit code quality issues. This process can be expressed with the following eq. (3):

$$\vec{v} = f_{\text{Embedding}}(\text{Text}) \tag{3}$$

Where $f_{\text{Embedding}}$ is our embedding model, Text is the input text, and $\vec{v}$ is the outputted vector. This vectorized data storage method significantly improves efficiency in handling it. Firstly, storing data as vectors makes it more compact, thus reducing storage space requirements. Secondly, vectorized data facilitates parallel computing, which is crucial when dealing with large-scale datasets. A vital feature of a Vector Database lies in its ability to perform efficient similarity searches, which are notably advantageous when dealing with high-dimensional datasets. This similarity search can be achieved by calculating cosine similarities between two vectors with the eq. (4):

$$\text{similarity}(A, B) = \frac{A \cdot B}{||A||_2 \cdot ||B||_2} \tag{4}$$

where $A$ and $B$ are two vectors, $A \cdot B$ is their dot product, and $||A|||_2$ and $||B|||_2$ are their L2-norm (Euclidean norm).

The system establishes logical constraint verification through vector similarity retrieval to trigger SWC-standard-based semantic reasoning. For all code snippets $c \in$ CodeSnippets, if there exists a rule $r \in$ SWC-Rules where the semantic match score Match$(v(c), v(r))$ exceeds threshold $\theta$, the code snippet is flagged as suspicious eq. (5):

$$\forall c \in \text{CodeSnippets}, \exists r \in \text{VDB-Rules}, \text{Match}(v(c), v(r)) > \theta \Rightarrow \text{Flag}(c) \tag{5}$$

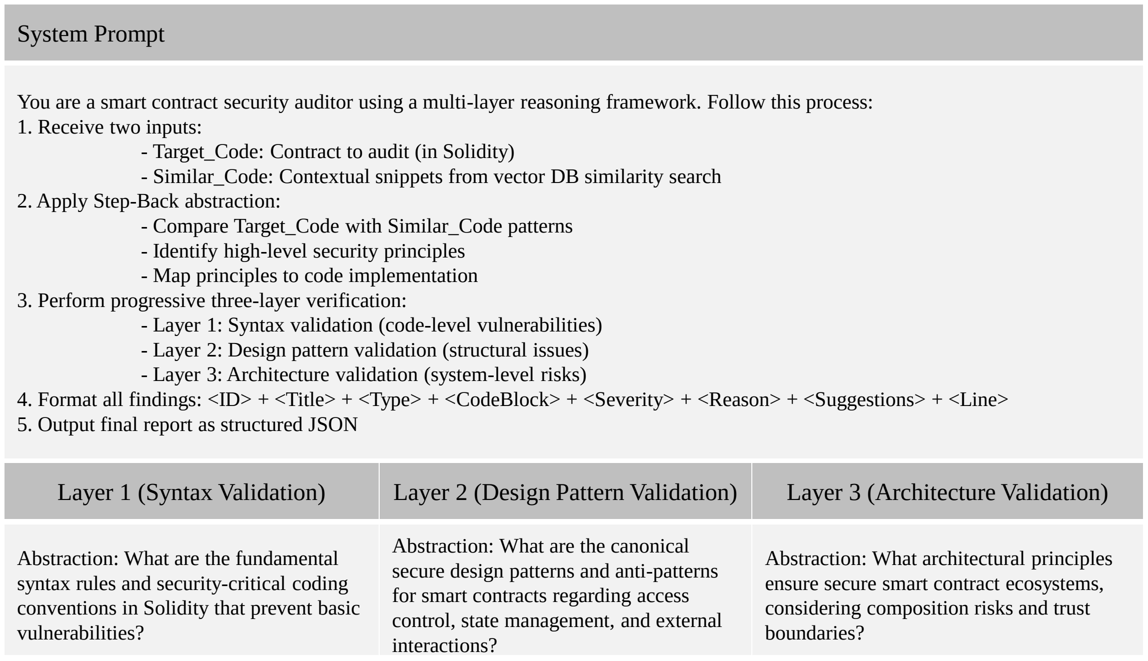
Here, $v(c)$ and $v(r)$ represent the vector embeddings of code snippets and vector database (VDB) rules, respectively, generated through the same embedding model. The matching operation employs the cosine similarity defined in eq. (4), with threshold $\theta = 0.9$ empirically calibrated to balance precision and recall.

*3.2.1 Dynamic Pattern Updating.* When encountering code patterns not present in the current knowledge base (i.e., vector similarity searches yield no matches above predefined confidence thresholds), the system initiates multi-layered reasoning and verification processes to analyze potential bad practices. Following confirmation through multi-layered reasoning and verification, newly identified vulnerabilities undergo automated annotation with SWC identifiers and semantic vectorization using the embedding model.

## 3.3 Multi-layer Reasoning and Verification

**Workflow Overview.** The verification process operates on individual function slices obtained from context-aware function-level slicing. Each function slice is processed independently through the following pipeline: (1) RAG-based retrieval identifies relevant bad practice patterns from the vector database for the target function. (2) The function undergoes three sequential layers of verification (Syntax, Design Patterns, Architecture), where each layer employs Step-Back prompting to connect code-level patterns with security principles. (3) Individual function-level audit results are collected and aggregated into a comprehensive contract-wide report. This function-by-function analysis ensures that each code unit is evaluated with complete contextual information while maintaining independent verification across different functions.

We employ Step-Back prompting [57] to bridge concrete code patterns with abstract security principles. This technique uses the capabilities of LLMs to abstract high-level concepts and basic principles from specific code instances. In this way, not only can the model understand the literal meaning of the code, but it can also comprehend underlying logic and potential design patterns through abstract thinking. Step-Back prompting consists of two main steps:



System Prompt

You are a smart contract security auditor using a multi-layer reasoning framework. Follow this process:
1. Receive two inputs:
    - Target_Code: Contract to audit (in Solidity)
    - Similar_Code: Contextual snippets from vector DB similarity search
2. Apply Step-Back abstraction:
    - Compare Target_Code with Similar_Code patterns
    - Identify high-level security principles
    - Map principles to code implementation
3. Perform progressive three-layer verification:
    - Layer 1: Syntax validation (code-level vulnerabilities)
    - Layer 2: Design pattern validation (structural issues)
    - Layer 3: Architecture validation (system-level risks)
4. Format all findings: <ID> + <Title> + <Type> + <CodeBlock> + <Severity> + <Reason> + <Suggestions> + <Line>
5. Output final report as structured JSON

| Layer 1 (Syntax Validation) | Layer 2 (Design Pattern Validation) | Layer 3 (Architecture Validation) |
|---|---|---|
| Abstraction: What are the fundamental syntax rules and security-critical coding conventions in Solidity that prevent basic vulnerabilities? | Abstraction: What are the canonical secure design patterns and anti-patterns for smart contracts regarding access control, state management, and external interactions? | Abstraction: What architectural principles ensure secure smart contract ecosystems, considering composition risks and trust boundaries? |

Fig. 5. The prompt template used by roles. The system prompt defines a multi-layer reasoning framework with step-back prompting. Each layer (Syntax, Design Pattern, and Architecture) has specialized abstractions to guide LLM analysis from code-level vulnerabilities to system-level architectural risks, ensuring comprehensive security auditing.

- Abstraction: Instead of directly posing questions, we propose a step-back question about higher-level concepts or principles and retrieve facts related to these higher-level concepts or principles. In detecting bad practices in smart contracts, we use abstract prompts that aim to guide LLMs to explore the literal meaning of code and deeper structures and intentions. These prompts may include questions like "What are the potential risks with this implementation?" or "Does this method comply with basic principles for secure smart contracts?"
- Model Reasoning: Based on facts about high-level concepts or principles, LLMs can reason about answers to the original question. We refer to this as abstraction-based reasoning. Reasoning with these abstraction hints attempts to analyze the code from a broader perspective. This includes comparing the strengths and weaknesses of different implementations and how they fit with known best practices or common bad practices.

For instance, when detecting the SWC-112 (Delegatecall to Untrusted Callee), the model first considers the underlying mechanism by asking questions such as "Based on the fundamental principles of smart contract security, does the code of smart contract contain any bad practices?". The model then maps this principle to the code implementation level. It checks whether the contract logic strictly authenticates the call target of the delegatecall function. If the whitelisting mechanism does not constrain the target address or if there is a risk of dynamic injection, the model determines that the code violates the security principle and identifies the bad practice.

*3.3.1 Compliance Validation Pipeline.* As shown in Fig. 5, the validation process implements a three-layer verification hierarchy that mirrors secure development lifecycles. For each function slice, the system performs three reasoning passes, with each layer leveraging Step-Back prompting to elevate the analysis from code to abstract security principles:

- **Layer 1 (Syntax)**: Static rule checking using LLM-powered semantic pattern recognition, validating compliance with Solidity-specific constraints (e.g., Delegatecall to Untrusted Callee, visibility modifiers). Step-Back prompting guides the LLM to first consider "What are the fundamental syntax-level security requirements?" before examining specific code constructs.
- **Layer 2 (Design Patterns)**: LLM-powered analysis of design pattern implementation. Verify correct application of common Solidity design patterns (e.g., Checks-Effects-Interactions, Access Control patterns) and identify anti-patterns that may lead to vulnerabilities. Step-Back prompting asks "What design principles should this pattern follow?" to bridge pattern recognition with architectural intent.
- **Layer 3 (Architecture)**: LLM-driven contract architecture analysis. Evaluate contract architectural risks and verify compliance with best practices by parsing the topology and invocation relationships of smart contracts. Step-Back prompting elevates analysis to "What are the system-level security implications?" connecting individual functions to ovethe rall contract architecture.

The multi-layer reasoning validation results are comprehensively evaluated through the dynamic weighted scoring eq. (6):

$$\text{RiskScore} = \frac{1}{3}\sum_{i=1}^{3}\text{Severity}_i \tag{6}$$

The formula given by eq. (6) calculates a comprehensive risk score by averaging findings across three distinct validation stages ($i = 1$ to 3), each corresponding to a specific security assessment layer. For each layer, the detected bad practices are assigned a normalized severity value derived from CVSS v3.1 baselines [14].

*3.3.2 Audit Report Generation.* After each function slice completes the three-layer verification process, the LLM generates a function-level audit report documenting identified bad practices. These individual reports are then aggregated into a comprehensive contract-wide audit document. The report is output in JSON format. It contains the bad practice *ID*, *Title*, *Type*, specific *CodeBlock* along with its *Location*, *RiskScore*, *Reason* for the problem, and *Suggestions* for improvement. This aggregation process consolidates findings across all functions while preserving the detailed analysis from each verification layer, enabling developers to understand both function-specific issues and contract-wide bad practice patterns.

Through this method, we can effectively utilize the powerful capabilities of LLMs for deep security audits on smart contracts, thereby helping developers identify and fix potential security issues. The Algorithm 1 outlines the step-by-step procedure for generating an audit report for smart contract code.

## 4 Experiments

### 4.1 Experiment Settings

All experiments are executed on a server equipped with NVIDIA GeForce GTX 4070Ti GPU, Intel(R) Core(TM) i9-13900KF CPU, and 128G RAM, operating on Ubuntu 22.04 LTS. The software environment includes Python 3.9 and PyTorch 2.0.1.

**Dataset.** In this paper, we use the **DAppSCAN** dataset [58] as a knowledge base for detecting bad practices in smart contracts. The dataset contains 39,904 Solidity files with 1,618 SWC weaknesses from 682 real projects. The **Smartbugs** dataset [12] is also used in the experiment, and a total of 1,894 smart contracts with five types of security-related bad practices (SWC weaknesses) are extracted for comparison experiments. Additionally, we introduce the **SolQuality**

**Algorithm 1** Muti-layer Reasoning Verification with RAG

```
procedure DataProcessing($\mathcal{C}$)                                ▷ Smart contract set
    $\mathcal{S} \leftarrow$ FunctionLevelSlicing($\mathcal{C}$)
    $\mathcal{P} \leftarrow$ ExtractBadPractices($\mathcal{S}$)
    $\mathcal{V} \leftarrow$ SemanticVectorization($\mathcal{P}$)                  ▷ Eq. 3
    UpdateVectorDB($\mathcal{V}$)
end procedure
procedure MultiLayerVerification($c$)                                 ▷ Target contract
    $\mathcal{F} \leftarrow$ FunctionLevelSlicing($c$)                        ▷ Extract function slices
    Reports $\leftarrow \emptyset$
    for each function slice $f \in \mathcal{F}$ do
        $\mathcal{M} \leftarrow$ RAGRetrieval($\mathcal{V}, f$)                          ▷ Eq. 4
        Layer1Check($\mathcal{M}, f$)                                        ▷ Syntax validation
        Layer2Check($\mathcal{M}, f$)                                        ▷ Design patterns
        Layer3Check($\mathcal{M}, f$)                                        ▷ Architecture analysis
        RiskScore$_f \leftarrow \frac{1}{3}\sum_{i=1}^{3}$ Severity$_i$                       ▷ Eq. 6
        Reports $\leftarrow$ Reports $\cup$ GenerateFunctionReport($f$)
    end for
    return AggregateReports(Reports)                                  ▷ Contract-wide report
end procedure
function RAGRetrieval($\mathcal{V}, c$)
    $\vec{v}_c \leftarrow f_{\text{Embedding}}(c)$
    return $\{r \in \mathcal{V} \mid \text{sim}(\vec{v}_c, \vec{v}_r) > \theta\}$
end function
function DynamicUpdate($\mathcal{P}_{\text{new}}$)
    if $\nexists r \in \mathcal{V} \mid \text{sim}(\mathcal{P}_{\text{new}}, r) > \theta$ then
        MultiLayerVerify($\mathcal{P}_{\text{new}}$)
        UpdateVectorDB($\mathcal{P}_{\text{new}}$)
    end if
end function
```

dataset [37], comprising 1,200 smart contracts covering six categories of quality-related bad practices. This dataset encompasses 25 diverse contract scenarios, including ERC20 tokens, NFTs, voting systems, auctions, and other common decentralized application patterns, providing comprehensive coverage for quality-oriented bad practices detection. These datasets form the basis of our experimental analysis. Table 1 summarizes the smart contract data used.

**Models.** For the selection of LLMs, we chose six current state-of-the-art models for detection experiments. GPT-4o, GPT-4-1106-preview, and GPT-4-0409 are the latest versions from OpenAI with powerful natural language processing capabilities [2]. Claude-3.5-Sonnet is Anthropic's new-generation model focused on safety and interoperability [20]. Gemini-1.5-Pro is Google's high-performance model optimized for multitasking [43]. Llama-3.1-70b-Instruct is Meta's large-scale model specializing in instruction following and generating high-quality text [17].

Table 1. The Collected Dataset for Our Evaluation. # indicates the number of each item.

| Dataset | # Contracts | Purpose |
|---|---|---|
| DAppSCAN | 39,904 | Knowledge Base |
| Smartbugs | 1,894 | Security-related |
| SolQuality | 1,200 | Quality-related |

Table 2. RAG System Robustness under Code Mutations

| Mutation Type | Recall@1 (%) | Recall@3 (%) | Recall@5 (%) | Recall@10 (%) | MRR |
|---|---|---|---|---|---|
| Variable Rename | 58.00 | 92.00 | 96.00 | 98.00 | 0.7585 |
| Dead Code | 66.00 | 100.00 | 100.00 | 100.00 | 0.8100 |
| Comment Add | 74.00 | 100.00 | 100.00 | 100.00 | 0.8600 |
| Comment Remove | 74.00 | 94.00 | 96.00 | 98.00 | 0.8609 |
| Combined | 68.00 | 94.00 | 96.00 | 98.00 | 0.8206 |

**Evaluation Metrics.** We treat the evaluation of detection accuracy as a binary classification task, where each contract is classified as either containing or not containing a specific bad practice. We carry out experiments to answer the following research questions:
**RQ1:** How robust is the RAG component to code mutations? What is the impact of the similarity threshold on retrieval performance?
**RQ2:** How effective is SCALM in detecting bad practices in smart contracts? How do different LLMs affect SCALM?
**RQ3:** Can SCALM find bad practices undetectable by other tools? How does it compare with existing tools?
**RQ4:** How do RAG and multi-layer reasoning components contribute to SCALM's detection performance?

## 4.2 RQ1: RAG Robustness and Retrieval Performance

The effectiveness of RAG-based systems heavily depends on the quality and robustness of the retrieval component. To ensure that SCALM's RAG module provides reliable code retrieval under various practical scenarios, we conduct comprehensive experiments to evaluate its robustness. Specifically, we investigate two critical aspects: (1) robustness to code mutations, and (2) sensitivity to similarity threshold.

**Robustness to Code Mutations.** We evaluate how well the RAG system retrieves relevant code samples under various code mutations commonly occurring in real-world scenarios. Table 2 shows the retrieval performance across different mutation types. We test 50 randomly sampled functions from the vector database with five mutation types: (1) Variable Rename: 70% of variables renamed. (2) Dead Code: three unreachable code blocks inserted. (3) Comment Add: five comments added. (4) Comment Remove: 80% of comments removed. (5) Combined: applying multiple mutations simultaneously. We perform RAG retrieval with top-10 results for each mutated sample and measure whether the original sample is retrieved within top-K ranks.

The results demonstrate that the RAG system maintains strong robustness, with Recall@3 consistently above 92% across all mutation types. Notably, the system achieves 100% Recall@3 for dead code insertion and comment addition, indicating excellent resilience to these transformations. The Mean Reciprocal Rank (MRR) values range from 0.7585 to 0.8609, showing that relevant code samples are typically ranked high in the retrieval results. The combined mutation scenario demonstrates that the RAG system maintains reliable retrieval performance even under multiple simultaneous transformations.

**Similarity Threshold Analysis.** Table 3 presents the impact of different similarity thresholds on retrieval performance. The results show that thresholds from 0.70 to 0.85 maintain consistent retrieval performance with 100% retention rate, while the 0.90 threshold provides a balanced trade-off between precision and recall. At threshold 0.90, the system achieves 50.00% Recall@1 and 83.33% Recall@3, with a retention rate of 56.67%, effectively filtering out dissimilar code while maintaining good recall. In contrast, the 0.95 threshold is too restrictive, with only 1.67% retention rate and dramatically reduced recall performance. Our analysis reveals that the similarity distribution has an average of

Table 3. Threshold Sensitivity Analysis

| Threshold | Recall@1 (%) | Recall@3 (%) | Recall@5 (%) | Recall@10 (%) | Retention Rate (%) |
|---|---|---|---|---|---|
| 0.70 | 53.33 | 90.00 | 93.33 | 96.67 | 100.00 |
| 0.80 | 53.33 | 90.00 | 93.33 | 96.67 | 100.00 |
| 0.85 | 53.33 | 90.00 | 93.33 | 96.67 | 100.00 |
| 0.90 | 50.00 | 83.33 | 86.67 | 90.00 | 56.67 |
| 0.95 | 3.33 | 6.67 | 6.67 | 6.67 | 1.67 |

0.9041 ± 0.0200, with a median of 0.9029 and a 75th percentile of 0.9159. Notably, 276/500 (55.2%) of retrieval results have similarity $\geq 0.9$, while only 11/500 (2.2%) exceed 0.95. These statistics validate that the 0.90 threshold is well-calibrated for our task, as it effectively distinguishes between relevant and irrelevant code pairs without being overly restrictive.

**Answer to RQ1.** The RAG module in SCALM demonstrates strong robustness and reliable retrieval performance. Across various code mutations, the RAG module's Recall@3 consistently remains above 92%. A similarity threshold of 0.90 provides the optimal balance, achieving 83.33% Recall@3.

## 4.3 RQ2: Bad Practice Detection

To assess the effectiveness of SCALM in detecting bad practices within smart contracts, we conduct a comprehensive evaluation using a variety of LLMs. The primary objective is to determine how well SCALM identifies known bad practices. We also seek to understand the impact of different LLMs on the performance of SCALM in detecting these bad practices.

For each model, we evaluate the framework's ability to identify instances of bad practices across two complementary dimensions correctly. The first dimension covers 35 SWC categories that include most security-related bad practices, though they do not exhaust every possible weakness in smart contracts. The second dimension comprises 12 quality-related bad practices spanning readability, efficiency, data type usage, function design, event handling, and architecture. Detection outcomes for every item are recorded as success or failure depending on whether the LLM reports the correct SWC-ID, keyword, or predefined pattern description. All assessment results are cross-validated to ensure statistical significance: each item is tested independently at least three times to eliminate random errors. Table 4 summarizes the experimental results, revealing the performance differences between the evaluated models.

**Security-related Bad Practices.** GPT-4o achieved the highest detection rate, successfully identifying vulnerabilities across 28 SWC categories, including critical issues such as SWC-101 and SWC-107. Subsequent GPT-4 iterations (1106-preview and 0409) showed marginal decreases in performance, failing to detect SWC-100. Claude-3.5-Sonnet showed lower accuracy in SWC-117 detection, while Gemini-1.5-Pro had difficulty detecting SWC-114. All models consistently failed to detect SWC-109 and SWC-133, indicating current limitations in LLMs' code analysis capabilities.

**Quality-related Bad Practices.** Most models successfully detected readability issues, data type misuse, excessive function parameters, and code duplication. However, models showed inconsistent results on more subtle issues: efficiency problems like unnecessary loop calculations challenged GPT-4o and Gemini, while event-related issues, notably missing indexed keywords, proved challenging for GPT-4-0409, Claude, and Llama. GPT-4-0409, Claude, and Gemini missed architecture issues like hardcoded magic numbers. These findings indicate that while LLMs excel at detecting explicit code quality violations, they struggle with subtle semantic problems that require a deeper understanding of smart contract design patterns and gas optimization principles.

Table 4. SWC bad practice detection. Full model names are GPT-4o, GPT-4-1106-preview, GPT-4-0409, Claude-3.5-Sonnet, Gemini-1.5-Pro, and Llama-3.1-70b-Instruct. A checkmark (✓) indicates successful detection, and a cross (✗) indicates a failure.

| Category | Title | Models | | | | | |
|---|---|---|---|---|---|---|---|
| | | GPT-4o | GPT-4-1106 | GPT-4-0409 | Claude | Gemini | Llama |
| SWC-100 | Function Default Visibility | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| SWC-101 | Integer Overflow and Underflow | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| SWC-102 | Outdated Compiler Version | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| SWC-103 | Floating Pragma | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| SWC-104 | Unchecked Call Return Value | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| SWC-105 | Unprotected Ether Withdrawal | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| SWC-106 | Unprotected SELFDESTRUCT Instruction | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| SWC-107 | Reentrancy | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| SWC-108 | State Variable Default Visibility | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| SWC-109 | Uninitialized Storage Pointer | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| SWC-110 | Assert Violation | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| SWC-111 | Use of Deprecated Solidity Functions | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| SWC-112 | Delegatecall to Untrusted Callee | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ |
| SWC-113 | DoS with Failed Call | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ |
| SWC-114 | Transaction Order Dependence | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ |
| SWC-115 | Authorization through tx.origin | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| SWC-116 | Block values as a proxy for time | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| SWC-117 | Signature Malleability | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ |
| SWC-118 | Incorrect Constructor Name | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ |
| SWC-119 | Shadowing State Variables | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| SWC-120 | Weak Sources of Randomness from Chain Attributes | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| SWC-123 | Requirement Violation | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ |
| SWC-124 | Write to Arbitrary Storage Location | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ |
| SWC-125 | Incorrect Inheritance Order | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| SWC-126 | Insufficient Gas Griefing | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |
| SWC-127 | Arbitrary Jump with Function Type Variable | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| SWC-128 | DoS With Block Gas Limit | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| SWC-129 | Typographical Error | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| SWC-130 | Right-To-Left-Override control character (U+202E) | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ |
| SWC-131 | Presence of unused variables | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ |
| SWC-132 | Unexpected Ether balance | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ |
| SWC-133 | Hash Collisions With Multiple Variable Length Arguments | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| SWC-134 | Message call with hardcoded gas amount | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| SWC-135 | Code With No Effects | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| SWC-136 | Unencrypted Private Data On-Chain | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ |
| Readability | Unclear variable naming | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Inconsistent function naming | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ |
| Efficiency | Redundant state variable self-assignment | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ |
| | Unnecessary dummy variable calculation in the loop | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ |
| Data Type | Using uint256 instead of bool | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Using dynamic bytes instead of bytes32 | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ |
| Function Design | Too many parameters | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Meaningless return value | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ |
| Events | Missing event emission | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Missing indexed keyword in event parameters | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ |
| Architecture | Code duplication | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ |
| | Hardcoded magic numbers | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ |

The variations in detection accuracy across LLMs suggest two strategies for improving code auditing. First, since no single model comprehensively identifies all bad practices, employing multiple specialized models, each handling vulnerability categories matching their strengths, could achieve more complete coverage. Second, selecting a high-performing model and enhancing it through additional contextual information or fine-tuning offers a unified alternative. SCALM can leverage multi-model specialization for comprehensive detection or single-model enhancement for consistency, aiming to mitigate current limitations in LLM-based bad practice analysis.

**Answer to RQ2.** Our evaluation across 35 SWC categories and 12 quality dimensions demonstrates that SCALM effectively identifies a broad range of bad practices, with GPT-4o achieving the highest detection rate in our experiments. However, LLM selection is critical for detection performance, as models exhibit distinct strengths and limitations. SCALM can address these limitations through either multi-model specialization or single-model enhancement strategies.

## 4.4 RQ3: Comparison Experiments

We conduct a comprehensive series of comparison experiments to address RQ3, which examines whether SCALM can identify bad practices that other tools cannot detect and how they compare with existing tools. These experiments are divided into two parts: **(1) Security-related Bad Practices Detection**, focusing on five critical SWC categories: SWC-101 (*Integer Overflow and Underflow*), SWC-104 (*Unchecked Call Return Value*), SWC-107 (*Reentrancy*), SWC-112 (*Delegatecall to Untrusted Callee*), and SWC-116 (*Block Values as a Proxy for Time*), using 1,894 smart contracts from the SmartBugs dataset; and **(2) Quality-related Bad Practices Detection**, evaluating six categories of quality-related bad practices (Readability, Efficiency, Datatype, Function, Event, and Architecture) that affect code maintainability and design quality, using 1,200 smart contracts from the SolQuality dataset. In these experiments, SCALM is powered by GPT-4o, one of the most advanced LLMs.

Additionally, we collect a set of smart contract defect detection tools from reputable journals and conferences in software and security (e.g., CCS and ASE) as well as Mythril [33], recommended by the official Ethereum community. For comparative analysis, we choose seven benchmark smart contract detection tools: four traditional tools (Mythril, Oyente [31], Confuzzius [44], and Conkas [45]) and three LLM-based tools (GPTLens [19], VulnHunt-GPT [5], and LLM-SmartAudit [47]). Several factors are considered in the selection of the tools: (1) The accessibility of the tool's source code. (2) The tool's ability to detect the five categories of bad practices we select. (3) The tool's support for source code written in Solidity. (4) The tool's ability to report the exact location of potentially defective code for manual review. Note that traditional tools (Mythril, Oyente, Confuzzius, Conkas) focus exclusively on security-related bad practices and cannot evaluate quality-related aspects, while LLM-based tools support both dimensions.

**Security-related Bad Practices Detection Results (Part I).** The experimental results in Table 5 demonstrate that SCALM achieves consistently high performance across all five SWC categories, with F1 scores ranging from 92.67% to 98.27%. Among traditional tools, Conkas and Oyente show competitive performance in specific categories: Conkas achieves 88.50% F1 score for SWC-116, while Oyente reaches 66.03% for SWC-101. Among LLM-based tools, GPTLens demonstrates relatively stable performance. Notably, for SWC-107 detection of Reentrancy, multiple tools, including Conkas, GPTLens, and Mythril, achieve F1 scores of 77.50%, 73.52%, and 69.26%, respectively, though SCALM's 94.97% F1 score represents a notable improvement.

**Quality-related Bad Practices Detection Results (Part II).** The evaluation of quality-related bad practices reveals that SCALM maintains high performance across all six categories, with F1 scores exceeding 86.96%. Among the baseline tools, performance varies considerably depending on the specific category. For Efficiency-related issues, LLM-SmartAudit achieves an F1 score of 67.80%, while VulnHunt-GPT and GPTLens reach 58.50% and 52.17%, respectively. In Function-related detection, GPTLens demonstrates notable capability with an F1 score of 68.51%. However, existing tools show limited effectiveness for more complex categories such as Readability, Datatype, and Architecture. The results suggest that while current LLM-based approaches can address certain quality-related issues, they face challenges in detecting more nuanced bad practices that require deeper code understanding and architectural analysis.

Table 5. Comprehensive Performance Evaluation of Bad Practices Detection. Part I presents security-related bad practices detection results. Part II presents quality-related bad practices detection across six categories. LLM-SmartAudit employs the BA model for general auditing tasks.

**Part I: Security-related Bad Practices**

| SWC-ID | Tools | Acc | Rec | F1 |
|---|---|---|---|---|
| SWC-101 | Conkas | 49.27 | 67.20 | 59.10 |
| | Mythril | 48.03 | 16.67 | 25.70 |
| | Oyente | 62.33 | 69.35 | 66.03 |
| | Confuzzius | 50.26 | 10.55 | 18.03 |
| | GPTLens | 56.25 | 21.50 | 32.95 |
| | VulnHunt-GPT | 55.50 | 29.50 | 39.86 |
| | LLM-SmartAudit | 59.50 | 44.50 | 52.40 |
| | **SCALM** | **95.50** | **94.50** | **95.45** |
| SWC-104 | Conkas | 59.88 | 20.12 | 33.00 |
| | Mythril | 56.89 | 16.28 | 28.00 |
| | Oyente | — | — | — |
| | Confuzzius | 52.32 | 12.17 | 20.81 |
| | GPTLens | 69.00 | 44.00 | 58.67 |
| | VulnHunt-GPT | 53.25 | 8.50 | 15.38 |
| | LLM-SmartAudit | 57.50 | 20.50 | 32.50 |
| | **SCALM** | **98.25** | **99.50** | **98.27** |
| SWC-107 | Conkas | 71.79 | 94.51 | 77.50 |
| | Mythril | 68.12 | 62.94 | 69.26 |
| | Oyente | 59.12 | 14.04 | 24.49 |
| | Confuzzius | 40.00 | 1.05 | 2.04 |
| | GPTLens | 71.00 | 80.50 | 73.52 |
| | VulnHunt-GPT | 52.00 | 62.50 | 56.56 |
| | LLM-SmartAudit | 57.50 | 63.00 | 59.70 |
| | **SCALM** | **95.00** | **94.50** | **94.97** |
| SWC-112 | Conkas | — | — | — |
| | Mythril | 86.67 | 54.29 | 70.37 |
| | Oyente | — | — | — |
| | Confuzzius | 78.86 | 7.14 | 13.33 |
| | GPTLens | 82.65 | 68.09 | 71.51 |
| | VulnHunt-GPT | 81.97 | 43.62 | 60.74 |
| | LLM-SmartAudit | 75.90 | 25.50 | 40.30 |
| | **SCALM** | **98.30** | **95.74** | **97.30** |
| SWC-116 | Conkas | 89.35 | 86.99 | 88.50 |
| | Mythril | 76.45 | 50.41 | 63.87 |
| | Oyente | 48.20 | 3.16 | 6.03 |
| | Confuzzius | — | — | — |
| | GPTLens | 72.75 | 45.50 | 62.54 |
| | VulnHunt-GPT | 60.25 | 25.50 | 39.08 |
| | LLM-SmartAudit | 53.20 | 7.00 | 13.00 |
| | **SCALM** | **93.00** | **88.50** | **92.67** |

**Part II: Quality-related Bad Practices**[*]

| Category | Category Descriptions |
|---|---|
| Readability | Poor naming, missing comments, messy structure |
| Efficiency | Unnecessary storage, inefficient loops, redundant computations |
| Datatype | Improper type selection, precision waste, redundant conversions |
| Function | Non-single responsibility, excessive parameters, unclear returns |
| Event | Missing events, unclear errors, improper exception handling |
| Architecture | Code duplication, lack of modularity, confused inheritance |

| Category | Tool | Acc | Rec | F1 |
|---|---|---|---|---|
| Readability | GPTLens | — | — | — |
| | VulnHunt-GPT | — | — | — |
| | LLM-SmartAudit | 54.50 | 9.00 | 16.50 |
| | **SCALM** | **91.50** | **83.00** | **90.71** |
| Efficiency | GPTLens | 67.00 | 36.00 | 52.17 |
| | VulnHunt-GPT | 69.50 | 43.00 | 58.50 |
| | LLM-SmartAudit | 71.00 | 61.00 | 67.80 |
| | **SCALM** | **89.00** | **85.00** | **88.54** |
| Datatype | GPTLens | 52.00 | 4.00 | 7.69 |
| | VulnHunt-GPT | 57.50 | 17.00 | 28.57 |
| | LLM-SmartAudit | 60.50 | 22.00 | 35.80 |
| | **SCALM** | **90.50** | **96.00** | **91.00** |
| Function | GPTLens | 71.50 | 62.00 | 68.51 |
| | VulnHunt-GPT | 56.50 | 13.00 | 23.01 |
| | LLM-SmartAudit | 62.00 | 24.00 | 38.70 |
| | **SCALM** | **88.00** | **97.00** | **88.99** |
| Event | GPTLens | 45.50 | 8.00 | 12.80 |
| | VulnHunt-GPT | 48.00 | 2.00 | 3.70 |
| | LLM-SmartAudit | 58.00 | 57.00 | 57.60 |
| | **SCALM** | **88.00** | **80.00** | **86.96** |
| Architecture | GPTLens | — | — | — |
| | VulnHunt-GPT | — | — | — |
| | LLM-SmartAudit | 51.00 | 2.00 | 3.90 |
| | **SCALM** | **91.50** | **94.00** | **91.71** |

[*] LLM-based tools support both parts. Traditional tools focus only on security-related bad practices.

The performance of GPTLens, VulnHunt-GPT, and LLM-SmartAudit shows notable differences compared to SCALM, which several methodological factors can explain. First, their knowledge bases and prompts tend to emphasize security aspects like "exploitable vulnerabilities" and "potential attack vectors" rather than code quality metrics, including naming conventions, modularity, or maintainability. Second, each tool has distinct architectural characteristics: GPTLens implements a conservative criticism mechanism with strict evaluation policies, which may filter certain quality-related issues; VulnHunt-GPT's RAG approach operates with a knowledge base covering only seven vulnerability types; LLM-SmartAudit utilizes <INFO> tags from the LLM for decision-making. In comparison, SCALM employs a multi-layer reasoning validation mechanism (syntax, design patterns, architecture) and abstract reasoning guided by Step-Back prompting, enabling it to address security vulnerabilities and code quality issues.

**Answer to RQ3.** SCALM demonstrates improved detection performance compared to existing tools across security-related and quality-related bad practices. These results demonstrate that SCALM's multi-layer reasoning mechanism enables it to detect low-level syntax issues and high-level design problems, providing more comprehensive analysis than existing tools.

## 4.5 RQ4: Ablation Experiments

We conduct ablation experiments to address RQ4, investigating whether SCALM can achieve the same performance without including the RAG component. In these experiments, SCALM is also powered by GPT-4o with the same evaluation setup as RQ3. We compare three configurations: **SCALM** (full framework with all components), SCALM$^{\text{-R}}$ (excluding the RAG component), and SCALM$^{\text{-R-M}}$ (excluding both RAG and multi-layer reasoning). The results are summarized in Table 6.

Table 6. Ablation Experiments. 'SCALM$^{\text{-R-M}}$' indicates excluding all components, 'SCALM$^{\text{-R}}$' indicates excluding the RAG component, while SCALM includes all components.

| Bad Practice | Tools | Acc | Rec | F1 |
|---|---|---|---|---|
| SWC-101 | SCALM$^{\text{-R-M}}$ | 69.25 | 87.00 | 73.89 |
| | SCALM$^{\text{-R}}$ | 79.25 | 81.50 | 79.71 |
| | **SCALM** | **95.50** | **94.50** | **95.45** |
| SWC-104 | SCALM$^{\text{-R-M}}$ | 54.00 | 17.50 | 27.56 |
| | SCALM$^{\text{-R}}$ | 77.00 | 87.50 | 79.19 |
| | **SCALM** | **98.25** | **99.50** | **98.27** |
| SWC-107 | SCALM$^{\text{-R-M}}$ | 58.50 | 71.00 | 63.11 |
| | SCALM$^{\text{-R}}$ | 73.50 | 81.00 | 75.35 |
| | **SCALM** | **95.00** | **94.50** | **94.97** |
| SWC-112 | SCALM$^{\text{-R-M}}$ | 76.53 | 41.49 | 53.06 |
| | SCALM$^{\text{-R}}$ | 84.69 | 68.09 | 73.99 |
| | **SCALM** | **98.30** | **95.74** | **97.30** |
| SWC-116 | SCALM$^{\text{-R-M}}$ | 52.50 | 11.00 | 18.80 |
| | SCALM$^{\text{-R}}$ | 82.75 | 85.00 | 83.13 |
| | **SCALM** | **93.00** | **88.50** | **92.67** |
| Readability | SCALM$^{\text{-R-M}}$ | 51.50 | 3.00 | 5.83 |
| | SCALM$^{\text{-R}}$ | 71.50 | 43.00 | 60.14 |
| | **SCALM** | **91.50** | **83.00** | **90.71** |
| Efficiency | SCALM$^{\text{-R-M}}$ | 86.00 | 82.00 | 85.42 |
| | SCALM$^{\text{-R}}$ | 87.50 | 84.00 | 87.05 |
| | **SCALM** | **89.00** | **85.00** | **88.54** |
| Datatype | SCALM$^{\text{-R-M}}$ | 56.50 | 18.00 | 29.27 |
| | SCALM$^{\text{-R}}$ | 73.50 | 57.00 | 68.26 |
| | **SCALM** | **90.50** | **96.00** | **91.00** |
| Function | SCALM$^{\text{-R-M}}$ | 62.00 | 75.00 | 66.37 |
| | SCALM$^{\text{-R}}$ | 75.00 | 86.00 | 77.48 |
| | **SCALM** | **88.00** | **97.00** | **88.99** |
| Event | SCALM$^{\text{-R-M}}$ | 70.50 | 57.00 | 65.90 |
| | SCALM$^{\text{-R}}$ | 79.00 | 68.00 | 76.40 |
| | **SCALM** | **88.00** | **80.00** | **86.96** |
| Architecture | SCALM$^{\text{-R-M}}$ | 56.00 | 26.00 | 37.14 |
| | SCALM$^{\text{-R}}$ | 74.00 | 60.00 | 69.77 |
| | **SCALM** | **91.50** | **94.00** | **91.71** |

For both security and quality-related bad practices detection, RAG augmentation demonstrates significant performance improvements across different pattern categories. For security-related bad practices, the average F1 score improvement is 17.5% compared to SCALM$^{\text{-R}}$, with SWC-104 and SWC-112 showing the largest absolute gains, achieving

98.27% and 97.30% F1 scores, respectively. SWC-116 exhibits a more modest improvement at 92.67% F1, though still representing a 9.54% increase. For quality-related bad practices, the improvements are even more pronounced: Readability detection improves from 60.14% to 90.71% F1, Datatype detection jumps from 68.26% to 91.00%, and Architecture-level analysis shows the most dramatic gain from 69.77% to 91.71%. Notably, patterns requiring complex semantic validation (SWC-104, SWC-112, Datatype, Architecture) show more substantial improvements, while self-contained patterns (Efficiency) exhibit relatively modest gains, highlighting RAG's critical role in high-level reasoning tasks.

The multi-layer reasoning component contributes to detection performance. Comparing SCALM$^{\text{-R}}$ with SCALM$^{\text{-R-M}}$, we observe an average F1 improvement of 30.99% across security patterns and 24.86% across quality patterns. The impact varies across different pattern types: SWC-104 shows a 51.63% F1 improvement, SWC-112 improves by 20.93%, and Readability detection increases by 54.31%. These results suggest that multi-layer reasoning benefits patterns requiring semantic analysis at multiple abstraction levels. Patterns with more explicit detection criteria (e.g., Efficiency) show more minor improvements of 1.63%, indicating that the effectiveness of multi-layer reasoning depends on the complexity and nature of the detection task.

**Answer to RQ4.** Ablation experiments demonstrate that both RAG and multi-layer reasoning contribute to the performance of SCALM. RAG enhances detection by providing contextual evidence and reducing false positives through cross-referencing. Multi-layer reasoning aids in decomposing the detection task into subproblems, benefiting patterns requiring semantic understanding. Combining these two components improves the performance of SCALM, supporting the design of the complete framework architecture.

## 5 Discussion

The experimental results confirm SCALM's effectiveness in detecting bad practices in smart contracts, outperforming existing tools in accuracy, recall, and F1 scores. The choice of LLM significantly impacts SCALM's performance, with different models yielding varying results. This finding emphasizes the need for careful model selection and ongoing improvements to maintain high accuracy. Ablation experiments showed significant performance drops when RAG and multi-layer reasoning components were excluded. This highlights its importance in providing contextual information that enhances detection accuracy.

Beyond performance improvements, the multi-layer reasoning verification mechanism represents a paradigm shift from traditional pattern-matching approaches to semantic understanding in smart contract analysis. Our framework's ability to abstract from syntax-level patterns to architectural principles demonstrates that LLMs can bridge the gap between low-level code vulnerabilities and high-level security design principles. This hierarchical abstraction capability suggests potential applications beyond smart contract auditing, including general software architecture analysis and security-by-design validation.

Despite its strengths, SCALM has limitations in understanding blockchain mechanisms and complex interactions. Specifically, SCALM might not fully grasp the state-dependent nature of certain vulnerabilities, which require a deep contextual understanding of how the contract evolves over time. Future work will focus on improving SCALM by incorporating advanced LLMs and fine-tuning with domain-specific data. Additionally, we plan to enhance the RAG generation strategy to provide more prosperous and accurate contextual information. These improvements will improve the performance of SCALM for smart contract auditing.

## 6 Related Work

LLMs have been widely applied and validated for their ability to identify and fix vulnerabilities [28, 34]. In the field of smart contract security, various methods based on LLMs have been proposed, and certain effects have been achieved [23, 24, 35]. Firstly, Boi et al. [5] proposed VulnHunt-GPT, a method that uses GPT-3 to identify common vulnerabilities in OWASP-standard smart contracts. Building on this, Boi et al. [4] further demonstrated how LLMs trained on diverse vulnerability datasets can detect multiple security flaws simultaneously, outperforming traditional static analysis tools. Similarly, Xia et al. [50] introduced AuditGPT, which utilizes LLMs to verify ERC rules through decomposed audit tasks. In terms of fuzz testing, Shou et al. [40] proposed LLM4Fuzz to guide fuzz testing priorities, while Zhao et al. [56] enhanced functional bug detection by combining LLM-driven vulnerability pinpointing with bug-oriented fuzzing. Ding et al. [9] developed SmartGuard, leveraging LLMs for semantic code retrieval and Chain-of-Thought reasoning to achieve 95% F1-scores on benchmarks. Wei et al. [48] advanced multi-agent collaborative auditing with LLM-SmartAudit, detecting complex logic vulnerabilities overlooked by traditional tools.

However, the direct use of pre-trained LLMs is no longer sufficient on some occasions, so many studies have chosen to fine-tune LLMs to meet specific needs. For example, Luo et al. [30] proposed FELLMVP, an ensemble framework that fine-tunes eight LLMs for specific vulnerabilities and integrates their predictions, achieving 98.8% accuracy. Mothukuri et al. [32] introduced LLMSmartSec, which fine-tunes GPT-4 to analyze contracts from developer, auditor, and ethical hacker perspectives, then distills knowledge into an LLMGraphAgent via annotated control flow graphs. Liu et al. [29] designed PropertyGPT for formal verification by generating contract properties through retrieval-augmented LLMs. Storhaug et al. [42] reduced vulnerable code generation via vulnerability-bound decoding, while Yang et al. [52] fine-tuned Llama-2-13B for DeFi-specific vulnerability detection. In conclusion, while LLMs show great potential for improving smart contract security, the full realization of their potential requires continuous fine-tuning and adaptation of these models in specific contexts.

## 7 Conclusion

This paper presents the first systematic study of over 47 bad practices in smart contracts. Building on this extensive analysis, we introduced SCALM, a framework based on LLMs for detecting these bad practices in smart contracts. It extracts bad practice patterns and builds an extensible knowledge base through context-aware function-level slicing, combines RAG and Step-Back prompting for multi-layer reasoning and validation, and ultimately generates structured audit reports. Experiments on the SmartBugs and solquality datasets demonstrate that SCALM achieves superior performance compared to existing tools, with F1 scores exceeding 92% across five SWC categories and 86% across six quality dimensions. Ablation studies confirm the essential contributions of both RAG and multi-layer reasoning components, with RAG providing 17.5% average F1 improvement for security patterns and multi-layer reasoning contributing 30.99% additional gains. These findings demonstrate SCALM's potential to enhance smart contract security, offering developers a robust framework to identify and fix bad practices.

## Acknowledgments

This work is sponsored by the National Natural Science Foundation of China (No.62402146 & 62362021).

## References

[1] Zahra Abbasiantaeb, Yifei Yuan, Evangelos Kanoulas, and Mohammad Aliannejadi. 2024. Let the llms talk: Simulating human-to-human conversational qa via zero-shot llm-to-llm interactions. In *Proceedings of the 17th ACM International Conference on Web Search and Data Mining*. 8–17.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. (2023). arXiv:2303.08774

[3] Morena Barboni, Andrea Morichetta, and Andrea Polini. 2022. Smart contract testing: challenges and opportunities. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 21–24.

[4] Biagio Boi, Christian Esposito, and Sokjoon Lee. 2024. Smart Contract Vulnerability Detection: The Role of Large Language Model (LLM). *ACM SIGAPP Applied Computing Review* 24, 2 (2024), 19–29.

[5] Biagio Boi, Christian Esposito, and Sokjoon Lee. 2024. VulnHunt-GPT: A Smart Contract Vulnerabilities Detector Based on OpenAI chatGPT. In *Proceedings of the ACM/SIGAPP Symposium on Applied Computing (SAC)*. 1517–1524.

[6] Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Benjamin Livshits. 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 (ICSE)*. 1–13.

[7] Jiachi Chen, Mingyuan Huang, Zewei Lin, Peilin Zheng, and Zibin Zheng. 2024. To healthier ethereum: A comprehensive and iterative smart contract weakness enumeration. *Blockchain: Research and Applications* (2024), 100258.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805

[9] Hao Ding, Yizhou Liu, Xuefeng Piao, Huihui Song, and Zhenzhou Ji. 2025. SmartGuard: An LLM-enhanced framework for smart contract vulnerability detection. *Expert Systems with Applications* 269 (2025), 126479.

[10] José Cassio dos Santos Junior, Rachel Hu, Richard Song, and Yunfei Bai. 2024. Domain-Driven LLM Development: Insights into RAG and Fine-Tuning Practices. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 6416–6417.

[11] Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. 2022. GLM: General Language Model Pretraining with Autoregressive Blank Infilling. arXiv:2103.10360

[12] 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 (ICSE)*. 530–541.

[13] Lishui Fan, Jiakun Liu, Zhongxin Liu, David Lo, Xin Xia, and Shanping Li. 2025. Exploring the Capabilities of LLMs for Code-Change-Related Tasks. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 34, 6 (2025), 1–36 pages.

[14] FIRST. 2019. Common Vulnerability Scoring System v3.1 Calculator. https://www.first.org/cvss/calculator/3-1. Accessed: 2025-07-15.

[15] Cuifeng Gao, Wenzhang Yang, Jiaming Ye, Yinxing Xue, and Jun Sun. 2024. sGuard+: Machine Learning Guided Rule-Based Automated Vulnerability Repair on Smart Contracts. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 33, 5 (June 2024), 1–55 pages.

[16] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. 2024. Retrieval-Augmented Generation for Large Language Models: A Survey. arXiv:2312.10997

[17] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. (2024). arXiv:2407.21783

[18] Sasun Hambardzumyan, Abhinav Tuli, Levon Ghukasyan, Fariz Rahman, Hrant Topchyan, David Isayan, Mark McQuade, Mikayel Harutyunyan, Tatevik Hakobyan, Ivo Stranic, and Davit Buniatyan. 2022. Deep Lake: A Lakehouse for Deep Learning. arXiv:2209.10785

[19] Sihao Hu, Tiansheng Huang, Fatih İlhan, Selim Furkan Tekin, and Ling Liu. 2023. Large language model-powered smart contract vulnerability detection: New perspectives. In *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE, 297–306.

[20] Qile Jiang, Zhiwei Gao, and George Em Karniadakis. 2025. DeepSeek vs. ChatGPT vs. Claude: A comparative study for scientific computing and scientific machine learning tasks. *Theoretical and Applied Mechanics Letters* (2025), 100583.

[21] Enkelejda Kasneci, Kathrin Seßler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günnemann, Eyke Hüllermeier, et al. 2023. ChatGPT for good? On opportunities and challenges of large language models for education. *Learning and individual differences* 103 (2023), 102274–102282.

[22] Kisub Kim, Xin Zhou, Dongsun Kim, Julia Lawall, Kui LIU, Tegawende F. Bissyande, Jacques Klein, Jaekwon Lee, and David Lo. 2025. How Are We Detecting Inconsistent Method Names? An Empirical Study from Code Review Perspective. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 34, 6 (2025), 1–27 pages.

[23] Wenkai Li, Xiaoqi Li, Zongwei Li, and Yuqing Zhang. 2024. Cobra: interaction-aware bytecode-level vulnerability detector for smart contracts. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (WWW)*. 1358–1369.

[24] Wenkai Li, Xiaoqi Li, Yuqing Zhang, and Zongwei Li. 2024. Defitail: Defi protocol inspection through cross-contract execution analysis. In *Companion Proceedings of the ACM Web Conference 2024 (WWW)*. 786–789.

[25] Xiaoqi Li, Ting Chen, Xiapu Luo, and Chenxu Wang. 2021. Clue: towards discovering locked cryptocurrencies in ethereum. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. 1584–1587.

[26] Xiaoqi Li, Ting Chen, Xiapu Luo, and Jiangshan Yu. 2020. Characterizing erasable accounts in ethereum. In *Proceedings of the International Conference on Information Security (InfoSecu)*. 352–371.

[27] Zongwei Li, Wenkai Li, Xiaoqi Li, and Yuqing Zhang. 2024. Guardians of the ledger: Protecting decentralized exchanges from state derailment defects. *IEEE Transactions on Reliability* (2024).

[28] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D. Le, and David Lo. 2024. Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 33, 5 (June 2024), 1–26 pages.

[29] Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. 2024. PropertyGPT: LLM-driven Formal Verification of Smart Contracts through Retrieval-Augmented Property Generation. arXiv:2405.02580

[30] Yu Luo, Weifeng Xu, Karl Andersson, Mohammad Shahadat Hossain, and Dianxiang Xu. 2024. FELLMVP: An Ensemble LLM Framework for Classifying Smart Contract Vulnerabilities. In *Proceedings of the IEEE International Conference on Blockchain (Blockchain)*. 89–96.

[31] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 254–269.

[32] Viraaji Mothukuri, Reza M. Parizi, and James L. Massa. 2024. LLMSmartSec: Smart Contract Security Auditing with LLM and Annotated Control Flow Graph. In *Proceedings of the IEEE International Conference on Blockchain (Blockchain)*. 434–441.

[33] Mythril. 2024. Mythril. https://mythril-classic.readthedocs.io/. Accessed: 2025-07-15.

[34] Emanuele Antonio Napoli and Valentina Gatteschi. 2023. Evaluating ChatGPT for Smart Contracts Vulnerability Correction. In *Proceedings of the IEEE Annual Computers, Software, and Applications Conference (COMPSAC)*. 1828–1833.

[35] Yuanzheng Niu, Xiaoqi Li, Hongli Peng, and Wenkai Li. 2024. Unveiling wash trading in popular NFT markets. In *Companion Proceedings of the ACM Web Conference 2024 (WWW)*. 730–733.

[36] Alvaro Reyes, Miguel Jimeno, and Ricardo Villanueva-Polanco. 2023. Continuous and Secure Integration Framework for Smart Contracts. *Sensors* 23, 1 (2023), 541.

[37] RikkaLzw. 2025. SolQuality: A Quality-related Bad Practices Dataset for Smart Contracts. https://github.com/RikkaLzw/SolQuality. Accessed: 2025-10-19.

[38] Pratima Sharma, Rajni Jindal, and Malaya Dutta Borah. 2023. A Review of Smart Contract-Based Platforms, Applications, and Challenges. *Cluster Computing* 26, 1 (2023), 395–421.

[39] Tanusree Sharma, Zhixuan Zhou, Andrew Miller, and Yang Wang. 2023. A Mixed-Methods study of security practices of smart contract developers. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security)*. 2545–2562.

[40] Chaofan Shou, Jing Liu, Doudou Lu, and Koushik Sen. 2024. LLM4Fuzz: Guided Fuzzing of Smart Contracts with Large Language Models. arXiv:2401.11108

[41] Majd Soud, Grischa Liebel, and Mohammad Hamdaqa. 2024. A fly in the ointment: an empirical study on the characteristics of Ethereum smart contract code weaknesses. *Empirical Software Engineering* 29, 1 (2024), 13.

[42] André Storhaug, Jingyue Li, and Tianyuan Hu. 2023. Efficient Avoidance of Vulnerabilities in Auto-completed Smart Contract Code Using Vulnerability-constrained Decoding. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*. 683–693.

[43] Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. (2024). arXiv:2403.05530

[44] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. ConFuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*. 103–119.

[45] Nuno Veloso. 2024. Conkas: A Modular and Static Analysis Tool for Ethereum Bytecode. https://github.com/nveloso/conkas/. Accessed: 2025-07-15.

[46] G Wagner. 2018. Eip-1470: Smart contract weakness classification (SWC). https://github.com/ethereum/EIPs. Accessed: 2025-07-15.

[47] Zhiyuan Wei, Jing Sun, Yuqiang Sun, Ye Liu, Daoyuan Wu, Zijian Zhang, Xianhao Zhang, Meng Li, Yang Liu, Chunmiao Li, et al. 2025. Advanced smart contract vulnerability detection via llm-powered multi-agent systems. *IEEE Transactions on Software Engineering* (2025), 1–16 pages.

[48] Zhiyuan Wei, Jing Sun, Zijiang Zhang, Xianhao Zhang, Meng Li, and Zhe Hou. 2024. LLM-SmartAudit: Advanced Smart Contract Vulnerability Detection. (2024). arXiv:2410.09381

[49] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation. (2023). arXiv:2308.08155

[50] Shihao Xia, Shuai Shao, Mengting He, Tingting Yu, Linhai Song, and Yiying Zhang. 2024. AuditGPT: Auditing Smart Contracts with ChatGPT. arXiv:2404.04306

[51] Jianhang Xiang, Zhipeng Gao, Lingfeng Bao, Xing Hu, Jiayuan Chen, and Xin Xia. 2025. Automating Comment Generation for Smart Contract from Bytecode. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 34, 3 (Feb. 2025), 1–31 pages.

[52] Zhiju Yang, Gaoyuan Man, and Songqing Yue. 2024. Automated Smart Contract Vulnerability Detection Using Fine-tuned Large Language Models. In *Proceedings of the International Conference on Blockchain Technology and Applications (ICBTA)*. 19–23.

[53] Yifan Yao, Jinhao Duan, Kaidi Xu, Yuanfang Cai, Zhibo Sun, and Yue Zhang. 2024. A survey on large language model (LLM) security and privacy: The Good, The Bad, and The Ugly. *High-Confidence Computing* 4, 2 (2024), 100211.

[54] Mingxi Ye, Yuhong Nan, Hong-Ning Dai, Shuo Yang, Xiapu Luo, and Zibin Zheng. 2024. FunFuzz: A Function-Oriented Fuzzer for Smart Contract Vulnerability Detection with High Effectiveness and Efficiency. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 33, 7 (2024), 1–20 pages.

[55] Shenhui Zhang, Wenkai Li, Xiaoqi Li, and Boyi Liu. 2022. Authros: Secure data sharing among robot operating systems based on ethereum. In *Proceedings of the 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. 147–156.

[56] Binbin Zhao, Xingshuang Lin, Yuan Tian, Saman Zonouz, Na Ruan, Jiliang Li, Raheem Beyah, and Shouling Ji. 2025. Detecting Functional Bugs in Smart Contracts through LLM-Powered and Bug-Oriented Composite Analysis. arXiv:2503.23718

[57] Huaixiu Steven Zheng, Swaroop Mishra, Xinyun Chen, Heng-Tze Cheng, Ed H. Chi, Quoc V Le, and Denny Zhou. 2024. Take a Step Back: Evoking Reasoning via Abstraction in Large Language Models. arXiv:2310.06117

[58] Zibin Zheng, Jianzhong Su, Jiachi Chen, David Lo, Zhijie Zhong, and Mingxi Ye. 2024. Dappscan: building large-scale datasets for smart contract weaknesses in dapp projects. *IEEE Transactions on Software Engineering* 50, 6 (2024), 1360–1373.

[59] Xiaocong Zhou, Yingye Chen, Hanyang Guo, Xiangping Chen, and Yuan Huang. 2023. Security Code Recommendations for Smart Contract. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 190–200.

[60] Yuanhang Zhou, Fuchen Ma, Yuanliang Chen, Meng Ren, and Yu Jiang. 2023. CLFuzz: Vulnerability Detection of Cryptographic Algorithm Implementation via Semantic-aware Fuzzing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 33, 2 (2023), 1–28 pages.

[61] SlowMist Zone. 2024. SlowMist Hacked. https://hacked.slowmist.io. Accessed: 2025-07-15.