SPEAR: An Engineering Case Study of Multi-Agent Coordination for Smart Contract Auditing

Indraveni Chebolu[0009−0008−5458−6429] ⋆, Arnab Mallick[0009−0009−6128−6253] ⋆⋆, and Harmesh Rana[0009−0000−7292−2029]

Centre for Development of Advanced Computing, Hyderabad, India
{indravenik,arnabm,harmeshr}@cdac.in
https://www.cdac.in

Abstract. We present SPEAR, a multi-agent coordination framework for smart contract auditing that applies established MAS patterns in a realistic security analysis workflow. SPEAR models auditing as a coordinated mission carried out by specialized agents: a Planning Agent prioritizes contracts using risk-aware heuristics, an Execution Agent allocates tasks via the Contract Net protocol, and a Repair Agent autonomously recovers from brittle generated artifacts using a programmatic-first repair policy. Agents maintain local beliefs updated through AGM-compliant revision, coordinate via negotiation and auction protocols, and revise plans as new information becomes available. An empirical study compares the multi-agent design with centralized and pipeline-based alternatives under controlled failure scenarios, focusing on coordination, recovery behavior, and resource use.

Keywords: Multi-agent systems · Smart contract auditing · Self-healing · Coordination protocols · Automated test repair

1 Introduction

The prevalence of decentralized applications, particularly in the Decentralized Finance (DeFi) [26] sector, has led to smart contracts that secure billions of dollars in value. Although this work unlocks a new financial model, it also introduces unprecedented risks, where a single vulnerability can lead to catastrophic financial losses, as exemplified by historic exploits such as the DAO hack [19]. Consequently, rigorous security auditing has become a fundamental necessity for the trust and stability of the blockchain ecosystem. However, the current audit paradigm, which relies heavily on expert manual analysis, creates a scalability bottleneck: audits typically require 2 - 4 weeks and cost $5,000 - $15,000 per project, failing to scale with the rapid pace of development in the Web3 space.

⋆ All authors contributed equally to this work.
⋆⋆ Corresponding author arnabm@cdac.in

To address this scalability issue, a variety of automated analysis tools have been developed. These tools, primarily based on static analysis and symbolic execution, have proven useful for identifying common vulnerability patterns. However, they suffer from several fundamental limitations that prevent them from serving as comprehensive audit solutions. First, they are overwhelmingly reactive, they scan for known patterns in isolated files but lack a holistic project-level understanding to strategically direct their analysis. Second, emerging generative tools for test creation are often brittle, they frequently produce code that fails to compile or run, and lack the mechanisms to autonomously recover from these errors, thus requiring frequent manual intervention. Finally, these tools are uncoordinated, operating as standalone programs without an overarching framework to coordinate their execution and synthesize their diverse output.

In this paper, the audit mission is the coordinated identification of state-reentrancy and access-control vulnerabilities across interdependent smart contract modules, together with the generation and validation of supporting evidence such as execution traces and tests. This mission is harder than a simple parallel search because findings in one module can change the risk assessment of another, while expensive and failure-prone analyzes must be scheduled across heterogeneous tools under limited budgets.

To overcome these limitations, we present SPEAR (Strategic Planning, Execution, and Automated Repair), a multi-agent coordination framework for smart contract auditing. SPEAR models auditing as a mission carried out by specialized agents (planning, execution, repair, safe actuation, and coordination) that maintain local beliefs, communicate via structured messages, and coordinate through explicit protocols (Contract Net, plan negotiation, resource auctions) to adapt plans when new information emerges. A multi-agent approach is essential because auditing is dynamic (new findings change priorities), resource-constrained (expensive operations must be budgeted), heterogeneous (different tools require different expertise) and requires coordination (failures must trigger replanning). We note that agents use deterministic policies rather than learning, the contribution is to demonstrate how established MAS coordination patterns enable adaptive auditing in this domain.

The primary contributions are:

1. an engineering case study demonstrating how established MAS coordination patterns can be applied to autonomous smart contract auditing.
2. a risk-aware Planning Agent that prioritizes audit tasks under changing information.
3. a Repair Agent implementing a programmatic-first repair policy for recovering from brittle generated artifacts.
4. a coordination framework that supports autonomous recovery and resource arbitration under failures, and
5. an empirical evaluation comparing a multi-agent design with centralized and pipeline-based alternatives, focusing on coordination and recovery behavior.

2 Background

Smart Contract Audit. Smart contracts are persistent, stateful programs that manage digital assets and interact through externally callable and cross-contract invocations. In DeFi deployments, a practical audit mission is not limited to scanning a single file, it requires reasoning about privilege boundaries, state-reentrancy, access-control decisions, asset-flow invariants, and interactions among interdependent modules that hold or move value[18, 26, 19]. In practice, auditors therefore combine complementary analysis modalities: static analysis to flag suspicious patterns quickly [9], symbolic execution to explore feasible paths [6], fuzzing to search for concrete failing executions [12], and proof-oriented analysis to establish critical properties against a formal semantics or specification [3, 13]. These tools are valuable in isolation but produce heterogeneous findings, consume different resources, and often require orchestration when applied to large contract sets.

Agent-Oriented Coordination. Multi-agent systems provide a natural model for such orchestration because different components can maintain local knowledge, pursue specialized goals, and coordinate through explicit communication [28, 20, 21]. In engineered MAS, agents usually coordinate through messages and standardized performatives such as INFORM, PROPOSE, ACCEPT, and REJECT [11, 2]. Task assignment can then be decoupled from task execution using coordination mechanisms such as the Contract Net Protocol [23], where agents advertise work, bid based on capability or availability, and accept awards dynamically [23]. This is well suited to auditing workflows in which priorities and tool availability change during execution.

Belief Revision and Self-Adaptation. Long-running audit workflows are failure-prone: generated artifacts may not compile, dynamic tools may time out, and new findings can invalidate earlier priorities. Self-adaptive systems address this through continual monitoring, analysis, planning, and execution over an evolving knowledge base [15, 27]. For knowledge maintenance, AGM theory offers a principled account of how an agent can revise beliefs when new evidence conflicts with prior assumptions, favoring minimal change while restoring consistency [1]. In an auditing setting, this matters because findings such as a newly confirmed vulnerability or an execution failure should update local reasoning without forcing a full restart of the mission.

Formal Verification as Complementary Evidence. Formal verification occupies a complementary place in the smart contract assurance landscape. Frameworks based on formal semantics and proof-oriented reasoning can establish strong guarantees about specified contract properties [3, 5, 13], whereas static analysis, symbolic execution, and fuzzing are typically used to surface actionable warnings or concrete counterexamples. In realistic audits, these techniques are often combined: proof-oriented methods provide high-assurance checks for critical properties, while search-based and execution-based methods help triage broader attack

surfaces and validate behaviors under concrete inputs. This complementarity motivates an orchestration layer that can schedule heterogeneous analyses, invoke formal methods when appropriate, and propagate their outcomes back into subsequent planning rather than assuming a single technique is sufficient.

3 Related Work

Smart Contract Analysis Tools. Automated smart contract analysis includes static analysis [9], symbolic execution [6], dynamic fuzzing [12], and formal verification [3]. More recent work applies large language models to vulnerability detection [30, 24] and repair [29, 14], mainly targeting production code. SmartAuditFlow [25] introduces planning-aware auditing, but does not address autonomous recovery from tool or artifact failures. In general, existing tools remain largely reactive and loosely integrated, requiring significant manual intervention and offering limited support for the coordinated use of heterogeneous analysis techniques.

Complementarity with Formal Verification. Formal verification tools such as the Certora Prover and K-based frameworks target exhaustive reasoning over explicitly specified properties [4, 5, 13, 3]. Their strength is proof-oriented assurance: they can prove or refute whether a formalized safety property holds for the analyzed contract model. SPEAR does not replace these solvers. Instead, it operates at the mission level, deciding when proof-oriented analysis should be invoked, how formal-verification results should be combined with static or dynamic findings, and how the audit should continue when tools fail or generated artifacts require repair. We therefore position SPEAR as a coordination layer around heterogeneous assurance techniques, including formal verification, rather than as a substitute for formal verification itself.

MAS Coordination Mechanisms. Multi-Agent Systems offer established coordination mechanisms for distributed problem solving. The Contract Net Protocol [23] supports the decentralized assignment of tasks based on agent capabilities, while auction-based mechanisms [22] enable the allocation of resources under partial information. BDI architectures [20, 21] provide a formal basis for modeling autonomous agents with beliefs, goals, and intentions. Previous MAS work in software engineering contexts [8] has focused primarily on distributed execution or runtime monitoring. In contrast, SPEAR applies these coordination patterns to orchestrate static and dynamic analysis tools with adaptive replanning and resource-aware decision making.

Self-Healing and Self-Adaptive Systems. Autonomic computing introduced the MAPE-K loop as a reference model for self-managing systems [15], and subsequent work on self-adaptive systems has explored mechanisms for runtime adaptation [27]. Program repair techniques range from search-based methods such as GenProg [16] to learning-based approaches, including Prophet [17] and

CodeBERT [10]. These methods primarily address defects in production code. By contrast, SPEAR focuses on repairing generated test artifacts that fail to compile or execute, using a programmatic-first repair policy to reduce reliance on costly generative invocations.

Multi-Agent Planning. Research on multi-agent planning studies how agents with partial and distributed knowledge coordinate to achieve shared objectives [7]. Distributed planning avoids the assumption of a complete global model and supports adaptation as local information changes. SPEAR does not employ classical planning formalisms such as PDDL or HTNs, instead, its Planning Agent uses heuristic risk-based prioritization and negotiation-driven replanning. This design aligns with multi-agent planning principles in which coordination emerges through local decision making and communication rather than centralized control.

4 The SPEAR Framework

SPEAR models auditing as a mission conducted by specialized agents that maintain local beliefs, communicate through structured messages, and coordinate through explicit protocols. As illustrated in Figure 1, the system contains five agent classes: the Planning Agent (A_P) constructs risk-aware audit plans, the Execution Agent (A_E) selects and schedules analysis tasks, the Repair Agent (A_R) repairs brittle generated artifacts, the Command Execution Agent (A_C) provides safe system actuation by sandboxing tool execution (Docker containers for Mythril, isolated environments for Echidna fuzz tests) and enforcing resource limits, and the Coordinator Agent (A_{Coord}) mediates conflicts and allocates shared resources. Global auditing behavior emerges from agent interactions rather than from a fixed pipeline.

Table 1 fixes the notation used throughout the paper. We reserve A_C for safe actuation and A_{Coord} for coordination. When multiple execution workers are instantiated, we write $A_{E,1}, A_{E,2}, \ldots$ for concrete instances of the Execution Agent class.

Table 1. Agent classes and notation used in SPEAR.

Symbol	Agent Class	Primary Responsibility
A_P	Planning Agent	Maintain risk scores and revise audit priorities
A_E	Execution Agent	Schedule analysis tasks and manage task allocation
A_R	Repair Agent	Repair brittle generated artifacts and report feasibility
A_C	Command Execution Agent	Execute tools safely under sandbox and budget limits
A_{Coord}	Coordinator Agent	Arbitrate conflicts and allocate shared resources

The key path through Figure 1 is that a security finding from A_E does not immediately trigger more tool calls. Instead, it first passes through AGM-compliant belief revision, updates A_P's beliefs, and can change the active plan before execution continues.

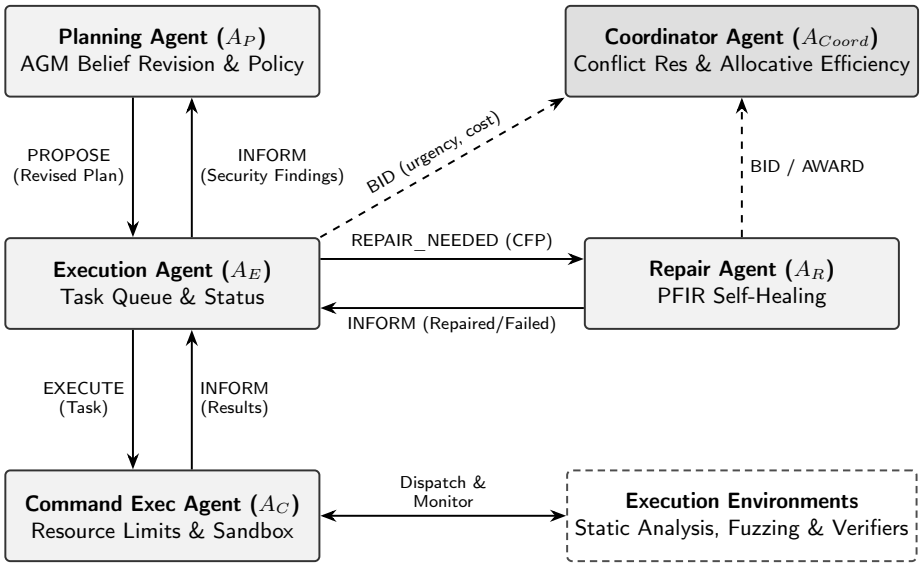

Fig. 1. *SPEAR architecture and coordination flow.* A security finding produced by A_E triggers AGM-compliant belief revision, updates the Planning Agent's beliefs, and can cause a revised plan to be negotiated before further tool execution.

Why Multi-Agent vs. Centralized? While a centralized controller could coordinate the same tools, MAS provides advantages under realistic failure and partial observability assumptions. We compare these architectural paradigms in Table 2 and demonstrate specific benefits through the following concrete scenarios:

Table 2. Architectural Comparison: SPEAR (MAS) vs. Traditional Approaches

Dimension	Centralized Pipeline	Stateless Microservices	SPEAR (MAS)
Fault Tolerance	Single point of failure	High (Retries)	**Graceful Degradation**
Decision Logic	Deterministic/Fixed	Event-driven	**Belief-based (AGM)**
Partial Failure	Sequence halts	Inconsistent state	**Autonomous Recovery**
Message Complexity	Low ($O(1)$)	Medium ($O(n)$)	**Dynamic Negotiation**
Adaptability	Requires restart	Config updates	**Real-time Replanning**

1. **Isolation of faults under partial observability**: *Scenario*: During audit, A_R repairs a contract test c_1 when a network partition isolates A_R from A_{Coord}. In a centralized system, the controller cannot receive repair status updates, causing it to time out and restart the repair from scratch (wasting the LLM budget). In SPEAR, A_R maintains local belief $B_R = \{\text{repairing}(c_1, \text{iteration} = 3), \text{last_fix}(\text{import_error})\}$ and continues repair autonomously. When connectivity resumes, A_R sends INFORM(repair_complete) and no work is lost. This property is necessary when agents have independent failure modes-realistic in distributed auditing where Mythril runs in Docker containers, Slither runs locally, and LLM calls traverse external APIs.

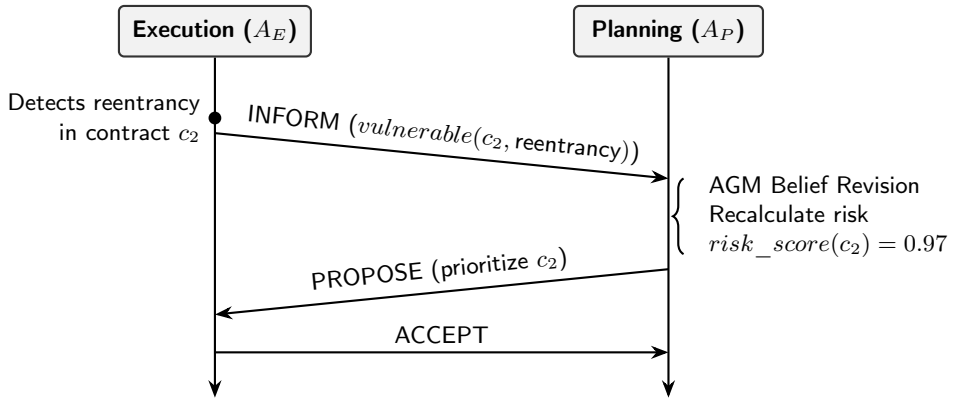

Fig. 2. *Plan Negotiation.* Distributed resolution of audit priorities without a central arbiter.

2. **Distributed Decision-Making Under Uncertainty**: *Scenario*: A_E discovers a reentrancy vulnerability in contract c_2, updating local beliefs: $B_E = \{\text{vulnerable}(c_2, \text{reentrancy}), \text{conf} = 0.9\}$. Simultaneously, A_P believes c_3 (a token contract with high TVL) has highest priority: $B_P = \{\text{risk_score}(c_3, 0.95)\}$. A centralized controller would need complete information about both agents' beliefs to arbitrate. In SPEAR, Plan Negotiation enables distributed resolution: A_E sends INFORM(reentrancy found in c_2); A_P updates B_P via belief revision, recalculating $\text{risk_score}(c_2) = 0.97 > 0.95$; A_P sends PROPOSE(prioritize c_2); A_E sends ACCEPT. Consensus emerges from local beliefs without a central arbiter, as illustrated in Figure 2.

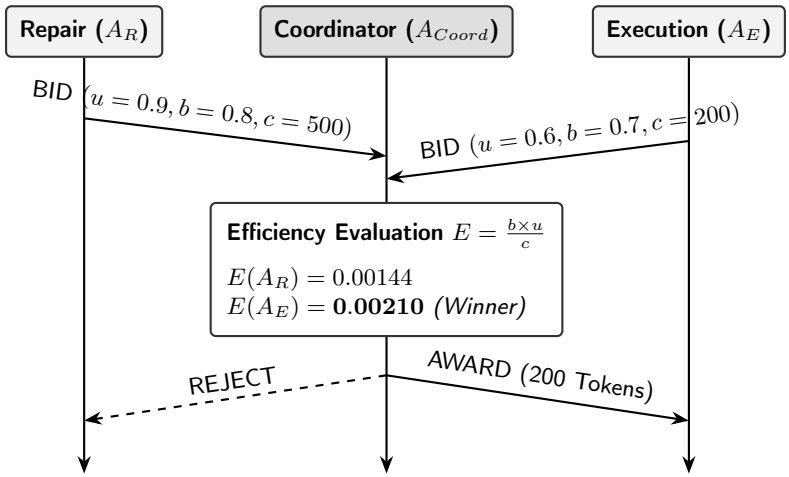

Fig. 3. *Resource Auction sequence.* The Coordinator agent evaluates concurrent bids and achieves allocative efficiency by awarding resources to the agent with the highest benefit-urgency to cost ratio.

3. **Resource Allocation Under Competing Interests**: *Scenario*: A_R needs 500 LLM tokens for complex repair, while A_E needs 200 tokens for test generation. Each agent knows its own urgency and expected benefit (private information). A centralized allocator would require both agents to report these values truthfully, creating incentive problems. In SPEAR, Resource Auction (Protocol 3) reveals true valuations through bidding: A_R bids $(\text{urgency} = 0.9, \text{benefit} = 0.8, \text{cost} = 500)$, A_E bids $(\text{urgency} = 0.6, \text{benefit} = 0.7, \text{cost} = 200)$. A_{Coord} computes the efficiency scores: $A_R = \frac{0.9 \times 0.8}{500} = 0.00144$, $A_E = \frac{0.6 \times 0.7}{200} = 0.0021$. Since $0.0021 > 0.00144$, A_E wins-the auction correctly allocates to the more efficient request (higher benefit per unit cost), even when urgency is lower. This shows how the mechanism elicits truthful bidding and achieves allocative efficiency (see Figure 3).

These scenarios illustrate how explicit coordination, local autonomy, and decentralized decision-making simplify recovery and adaptation under partial observability and independent failure modes. We acknowledge that MAS is sufficient rather than strictly necessary for adaptive auditing-a sophisticated centralized system with modular components could implement similar logic.

We do not claim incentive compatibility, the auction serves as a lightweight heuristic for prioritizing competing requests among cooperative agents rather than as a mechanism with dominant-strategy guarantees.

4.1 Formal Model and Coordination

We model SPEAR as a multi-agent system $\mathcal{M} = (\mathcal{A}, \mathcal{S}, \mathcal{E}, \mathcal{C})$ where $\mathcal{A} = \{A_P, A_E, A_R, A_C, A_{Coord}\}$ is the set of agents, $\mathcal{S}$ is the state space of the system, $\mathcal{E}$ is the set of events and $\mathcal{C}$ is the set of coordination protocols. We now formally define each component.

System State $\mathcal{S}$. The state of the system $s \in \mathcal{S}$ is a tuple $s = (\mathcal{C}ont, \mathcal{T}ools, \mathcal{V}uln, \mathcal{R}es, \mathcal{M}sg)$ where:

- $\mathcal{C}ont = \{c_1, \ldots, c_n\}$ is the set of audit contracts, each c_i having attributes $\text{complexity}(c_i) \in [0, 1]$, $\text{dependencies}(c_i) \subseteq \mathcal{C}ont$, and $\text{test_coverage}(c_i) \in [0, 1]$.
- $\mathcal{T}ools = \{\text{Slither}, \text{Mythril}, \text{Echidna}, \ldots\}$ is the set of available analysis tools.
- $\mathcal{V}uln = \{v_1, \ldots, v_m\}$ is the set of vulnerabilities discovered, each v_i having attributes $\text{severity}(v_i) \in \{\text{critical}, \text{high}, \text{medium}, \text{low}\}$, $\text{contract}(v_i) \in \mathcal{C}ont$, and $\text{status}(v_i) \in \{\text{detected}, \text{confirmed}, \text{repaired}\}$.
- $\mathcal{R}es = (\text{llm_budget}, \text{time_budget}, \text{compute_budget})$ represents resource constraints, where each budget is a non-negative real number.
- $\mathcal{M}sg$ is the message queue containing pending inter-agent messages.

Events $\mathcal{E}$. Events $e \in \mathcal{E}$ represent state changes and include:

- $\text{VULN_DETECTED}(v, c)$: Vulnerability v detected in contract c.
- $\text{RISK_CHANGED}(c, \Delta)$: Risk assessment for contract c changed by Δ.

- TOOL_FAILED(t, error): Tool t failed with error message.
- REPAIR_NEEDED(artifact, failure_type): The generated artifact requires repair.
- RESOURCE_REQUESTED(agent, resource_type, amount): Agent requests resource allocation.
- TIMEOUT(protocol_id): Protocol timeout event.

Events trigger agent perception and may cause state transitions: $\delta : \mathcal{S} \times \mathcal{E} \to \mathcal{S}$.

Agent Model. Each agent $A_i \in \mathcal{A}$ is a tuple $A_i = (B_i, G_i, \pi_i, I_i)$:

- **Beliefs** B_i: $B_i \subseteq \mathcal{L}$ where $\mathcal{L}$ is a first-order language with predicates for vulnerabilities, risk scores, tool states, and resource availability. Each belief has confidence $\text{conf}(\phi) \in [0, 1]$. The revision of beliefs follows the AGM postulates [1] with minimal confidence-weighted change: when new evidence conflicts with existing beliefs, the beliefs of the lowest-confidence are first removed to restore consistency.
- **Goals** G_i: Prioritized formulas representing the desired states. Conflicting goals are resolved by selecting the highest-priority goal.
- **Policy** π_i: $\pi_i : B_i \times G_i \to \mathcal{A}ct$ maps beliefs and goals to actions. Policies are deterministic (not learned).
- **Intentions** I_i: Goals committed that agents pursue until they are achieved or impossible. Intention persistence ensures that agents complete ongoing tasks before responding to new priorities.

Actions and Execution. Actions include communication primitives (INFORM, PROPOSE, ACCEPT, REJECT), coordination primitives (CFP, BID, AWARD), and domain actions (EXECUTE, REPAIR). Each action has preconditions and effects on the state of the system. EXECUTE and REPAIR actions may succeed or fail, failures trigger TOOL_FAILED or REPAIR_NEEDED events. Agents execute in perceive-decide-act cycles, processing messages via FIFO queues.

Agent Execution Semantics. In the implementation, the agents run asynchronously with a central message broker. Agents maintain a local state during partitions and resynchronize when connectivity resumes.

1. **Perceive**: Agent A_i receives messages $M = \{m_1, \ldots, m_k\}$ from its FIFO message queue, processing in arrival order. For each m_j, update beliefs: $B_i \leftarrow U_i(B_i, m_j)$.
2. **Decide**: Select active goal $g^* = \text{resolve}(G_i \cap I_i)$ (or $g^* = \text{resolve}(G_i)$ if $I_i = \emptyset$). Compute the action $a = \pi_i(B_i, \{g^*\})$. If a is executable, commit to intention: $I_i \leftarrow I_i \cup \{g^*\}$.
3. **Act**: Execute action a, updating the state of the system: $s \leftarrow \text{eff}(a)(s)$. If $B_i \models g^*$, remove from intentions: $I_i \leftarrow I_i \setminus \{g^*\}$.

If multiple agents attempt conflicting actions in the same round (e.g., both request the same resource), A_{Coord} arbitrates via the resource auction protocol. This cycle repeats until all goals are achieved or the system is terminated.

Coordination Protocols. SPEAR uses three coordination protocols, summarized in Table 3. Each is a finite state machine with timeout-based termination to ensure deadlock-freedom.

Table 3. Coordination Protocols Summary

Protocol	Purpose	Key Messages	Timeout
Plan Negotiation	Priority coordination	INFORM, PROPOSE, ACCEPT/REJECT	30s
Contract Net	Task allocation	CFP, BID, AWARD	10s
Resource Auction	LLM budget allocation	OPEN, BID, ALLOCATE	15s

Plan Negotiation coordinates between A_P and A_E when new findings change risk assessments. A_E sends INFORM with the findings, if the risk change exceeds the threshold $\theta = 0.15$, A_P proposes the revised plan. The negotiation history is bounded ($H_{\max} = 5$) to prevent infinite loops, and unresolved negotiations escalate to A_{Coord}.

Contract Net [23] assigns analysis tasks. A_E broadcasts the CFP, agents bid based on capability and availability (score $= 0.7 \cdot$capability$+0.3 \cdot$availability), and the highest-scoring agent is awarded the task.

The resource auction allocates scarce resources (LLM budget). Agents bid with (urgency, benefit, cost), allocation maximizes efficiency: $\frac{\text{urgency} \times \text{benefit}}{\text{cost}}$.

System Properties. We state properties with proof sketches.

Safety (Deadlock-Freedom): $\forall s \in \mathcal{S}, \exists e \in \mathcal{E}$ such that $\delta(s, e) \neq s$.

Sketch: Protocol timeouts ensure that no agent blocks indefinitely. If protocols are in accepting states but goals remain, agents can initiate new protocols (no preconditions on CFP, PROPOSE). Circular waits are broken down by timeout events. □

Conditional Progress (Task Progress): Under assumptions (A1) all contracts have positive risk scores, and (A2) tool completeness, audit tasks eventually make progress.

Sketch: By A1, all contracts appear in the A_P' plan. By termination of Protocol 2, all contracts are analyzed. By A2 and the soundness of the tool, detected vulnerabilities are real. □

Termination: All protocols end within $T_{\max} \leq 30$s (individual) or ≤ 45s (nested).

Sketch: History bounds ($H_{\max} = 5$) and timeouts ensure bounded traces. □

Consistency: Belief revision maintains consistency via AGM-compliant minimal change. □

4.2 The Strategic Planning Agent

The Planning Agent A_P maintains and continuously reviews a risk-aware audit policy, demonstrating key properties of MAS agents:

Autonomy: A_P makes independent decisions through policy π_P based on local beliefs B_P about contract risks, without requiring external directives. Its

goal $G_P = \{\text{all_contracts_audited}, \text{critical_vulns_found_first}\}$ drives the generation of autonomous plans.

Proactivity: A_P initiates plan revisions when beliefs change (via belief revision U_P), even without explicit requests. When A_E sends INFORM(findings) and $|\Delta_{\text{risk}}| > \theta$, A_P proactively sends PROPOSE(revised plan) to renegotiate priorities.

Reactivity: A_P responds to events (vulnerability detections, risk changes) by updating beliefs and revising plans within the perceive-decide-act cycle.

Social Ability: A_P coordinates through the plan negotiation protocol (Section 3.1) with A_E, exchanging INFORM and PROPOSE messages to reach agreement on audit priorities.

A_P uses a greedy heuristic prioritizing contracts by risk score: $\text{risk_score}(c) = \alpha \cdot \text{complexity}(c) + \beta \cdot \text{dependency_risk}(c) + \gamma \cdot \text{test_coverage_risk}(c)$ ($\alpha = 0.5$, $\beta = 0.3$, $\gamma = 0.2$), running in $O(|C| \log |C|)$ time and empirically improves early discovery. Policy π_P maps beliefs about contract risks to plan generation actions, autonomously revising plans when the change in risk assessment exceeds the threshold $\theta = 0.15$.

4.3 The Repair Agent

The Repair Agent A_R diagnoses and repairs damaged generated artifacts (primarily test code), demonstrating the properties of the MAS agent.

Autonomy: A_R makes independent repair decisions via policy π_R based on local beliefs B_R about artifact failures and repair strategies. Its goal $G_R = \{\text{artifact_repaired}, \text{minimize_llm_cost}\}$ drives the selection of the autonomous repair strategy.

Proactivity: A_R initiates repair attempts when it receives REPAIR_NEEDED events, without waiting for explicit commands. proactively selects repair strategies (deterministic vs. generative) based on failure patterns in beliefs.

Reactivity: A_R responds to repair failures by updating beliefs about what strategies work, adapting its policy π_R accordingly within the perceive-decide-act cycle.

Social Ability: A_R coordinates through the Contract Net protocol (Section 3.1) to bid on repair tasks and communicates repair results via INFORM(repair_result) messages to A_E and A_P.

A_R applies a programmatic-first repair policy that prioritizes deterministic fixes before escalating to generative repair, reducing the dependence on costly external calls. The policy treats an artifact as *worth fixing* when the failure is local and mechanically diagnosable (e.g., missing imports, wrong identifiers, malformed assertions, or harness configuration errors), the underlying security objective still targets a high-priority contract, and the estimated repair effort fits within the remaining repair budget. Under those conditions, A_R first applies deterministic templates and only escalates to a generative repair step when the artifact remains mission-critical after local fixes fail. By contrast, the mission is replanned instead of repeatedly repairing when failures are environmen-

tal or semantically non-local (e.g., repeated tool crashes, unavailable dependencies, incompatible assumptions about contract state), when PFIR reaches MAX_ATTEMPTS (typically 5), or when the expected repair cost exceeds the value of continuing the current objective relative to higher-risk pending tasks. In that case, A_R sends FAILURE(failure_class, attempts, cost) to A_E and A_P, allowing the system to update beliefs about repair feasibility and switch to a revised plan.

5 Experiments and Evaluation

We conducted a series of experiments to empirically evaluate the performance of the SPEAR framework. Our evaluation is designed to answer four central research questions (RQs) that assess the framework's effectiveness, robustness, efficiency, and MAS-specific properties (autonomy, coordination, communication).

Statistical Methodology: The experiments were repeated in multiple independent runs. We report mean values with standard deviation to capture variability.

5.1 RQ1: How does coordination structure affect audit task progress under identical analysis logic?

Experimental Setup. We evaluated SPEAR on the Damn Vulnerable DeFi benchmark (15 challenges, 10 runs per configuration). Baselines:

1. **Slither only**: static analysis without coordination,
2. **Sequential Pipeline**: Slither→Mythril→Echidna executed sequentially without coordination, failure recovery, or adaptive planning,
3. **Slither+Mythril**: combined static and symbolic analysis
4. **Centralized Scheduler**: a sophisticated baseline that implements the same risk-aware planning heuristics and PFIR repair logic as SPEAR, but using a centralized architecture. The Centralized Scheduler consists of: (a)
   (a) a priority queue ordered by risk score ($\alpha = 0.5, \beta = 0.3, \gamma = 0.2$)
   (b) event handlers for tool completion/failure
   (c) modular repair component with programmatic-first strategy, and
   (d) single-threaded controller that polls for events and dispatches tasks
   This baseline isolates the contribution of MAS coordination, it has identical logic but lacks distributed agents, negotiation protocols, and auction-based resource allocation. Ground truth: known vulnerabilities in the DVD repository. Metrics: Precision, Recall, F1-Score.

Results. Table 4 summarizes the effectiveness results. SPEAR achieves higher overall effectiveness than all baselines, including a centralized scheduler that implements identical planning and repair logic. While centralized coordination performs comparably under normal conditions, the multi-agent design exhibits lower variance and more stable progress under injected failures (see RQ4).

Table 4. Effectiveness of SPEAR vs. Baselines on the Damn Vulnerable DeFi Dataset (15 challenges).

Approach	Precision	Recall	F1-Score
Slither only	0.82	0.65	0.73
Slither + Mythril	0.84	0.72	0.78
Sequential Pipeline	0.86	0.78	0.82
Centralized Scheduler	0.85	0.81	0.83
SPEAR (Full Framework)	**0.89**	**0.85**	**0.87**

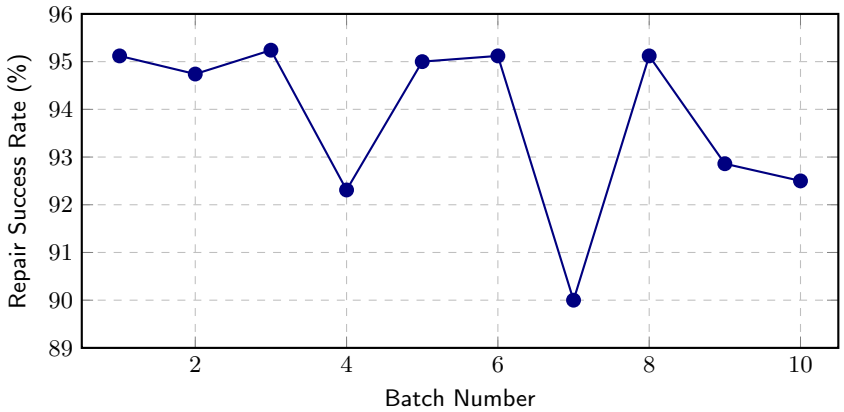

Fig. 4. *Performance of the PFIR self-healing algorithm* The programmatic-first echelon resolves most failures, achieving consistent success across batches.

5.2 RQ2: How effective is the PFIR self-healing algorithm?

Experimental Setup. We evaluate PFIR on 500 security objectives across open-source Solidity contracts. We compare PFIR with baselines: (1) Retry-only (regenerate up to 5 times) (2) LLM-only (always use generative repair). Repair success: the test compiles, executes, and triggers the security objective.

Results. Figure 4 summarizes the repair results. PFIR consistently recovers from generated artifact failures across batches, with most repairs handled deterministically before escalating to generative repair. This behavior reduces dependence on external calls while ensuring progress under repeated failures.

5.3 RQ3: Does the Strategic Planning Agent improve audit efficiency?

Experimental Setup. To measure the impact of the strategic planning agent, we conducted an ablation study on a DeFi protocol (47 contracts, 12 vulnerabilities). We ran the system in two modes: (1) **Full SPEAR**: complete framework with Planning Agent, (2) **Ablated SPEAR**: Planning Agent disabled, contracts analyzed in alphabetical order. Our hypothesis is that the planning-driven mode finds high-impact vulnerabilities more quickly.

Metric. Time to First Critical Vulnerability (TFCV): wall-clock time until the first critical vulnerability is identified.

Results. Figure 5 compares planning-driven and ablated execution. The planning agent enables earlier discovery of high-risk vulnerabilities by prioritizing contracts based on evolving risk assessments rather than static ordering.

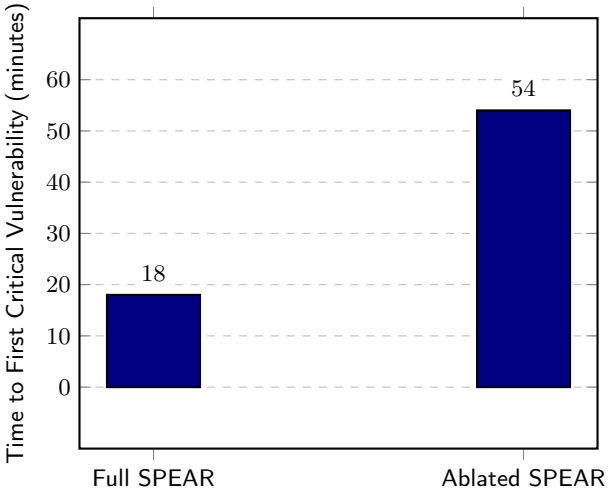

Fig. 5. *Impact of Strategic Planning Agent on Audit Efficiency.* The planning-driven mode detects critical vulnerabilities significantly faster than the ablated baseline. Horizontal grid lines indicate 10-minute intervals for clearer scale reference.

Note on Comparisons: Prior work on smart contract repair (e.g., SmartFix achieving ≈94.8% for vulnerability patching) addresses a different task: repairing production code vulnerabilities. PFIR addresses generated test code failures (compilation errors, missing imports, incorrect assertions). Direct comparison is not meaningful, we report PFIR's 94% success rate as evidence of effectiveness for the test repair task, with 64% deterministic fixes reducing LLM costs.

5.4 RQ4: How do MAS components (protocols, autonomy, self-healing) contribute to system robustness?

Experimental Setup. We conducted systematic ablation studies to isolate MAS contributions across 20 audit sessions, injecting controlled failures (tool crashes, network timeouts, LLM API failures). We compare five configurations:

- Full SPEAR: complete MAS framework
- No Protocols: agents operate autonomously but use direct method calls instead of protocols (no Contract Net, plan negotiation, or auctions)
- No Autonomy: agents follow fixed policies without local decision-making (centralized controller with same logic)

- No Self-Healing: repair agent disabled, failures require manual intervention
- Rigid Pipeline: sequential execution, no coordination, no autonomy, no self-healing. We measure recovery time, resource efficiency, coordination overhead, and resilience to failure.

Results. Table 5 summarizes the ablation study isolating coordination protocols, agent autonomy, and self-healing behavior under injected failures.

Protocol Contribution: Comparison of Full SPEAR vs. No Protocols shows that protocols enable 1.8× faster recovery (2.3 min vs. 4.1 min, $p < 0.01$) and 15% reduction in LLM invocations (1.5 vs. 1.76, $p < 0.05$). Protocols enable distributed decision-making with partial observability, allowing agents to negotiate priorities without a central arbitrator.

Autonomy Contribution: Comparing Full SPEAR vs. No Autonomy shows that autonomy enables 2.1× faster recovery (2.3 min vs. 4.8 min, $p < 0.01$) and 18% reduction in LLM invocations (1.5 vs. 1.83, $p < 0.05$). Local decision-making allows agents to adapt to failures without waiting for the central controller.

Self-Healing Contribution: Comparing Full SPEAR vs. No Self-Healing shows that self-healing enables 3.2× faster recovery (2.3 min vs. 7.4 min, $p < 0.001$) and 31% reduction in LLM invocations (1.5 vs. 2.17, $p < 0.001$). Autonomous repair prevents cascading failures.

Coordination Overhead: Full SPEAR has 4.2% overhead (std: 1.1%), with a mean of 47 messages per audit. Protocol breakdown: Plan Negotiation (1.8%), Contract Net (1.5%), Resource Auction (0.9%). The overhead scales linearly with the number of agents: $O(|\mathcal{A}|)$ messages per protocol instance.

Table 5. Ablation study of coordination protocols, agent autonomy, and self-healing under injected failures (20 audit runs).

Metric	Full SPEAR	No Protocols	No Autonomy	No Self-Healing	Rigid Pipeline
Recovery Time (min)	**2.3 ± 0.8**	4.1 ± 1.2	4.8 ± 1.5	7.4 ± 2.3	8.7 ± 2.1
LLM Invocations/Repair	**1.5 ± 0.4**	1.76 ± 0.5	1.83 ± 0.5	2.17 ± 0.6	1.95 ± 0.5
Negotiation Success Rate	**94%**	N/A	N/A	94%	N/A
Conflict Frequency/Audit	**1.2 ± 0.6**	2.8 ± 1.1	0	1.2 ± 0.6	0
Messages/Audit	**47 ± 8**	12 ± 3	0	47 ± 8	0
Coordination Overhead	**4.2% ± 1.1%**	1.8% ± 0.5%	0%	4.2% ± 1.1%	0%

The ablation results indicate distinct roles for coordination protocols, agent autonomy, and self-healing behavior under injected failures.

5.5 Threats to Validity

Internal Validity: Controlled failure injection (RQ4) may not capture all real-world failure modes. Ablation comparisons assume implementations are equivalent except for the ablated component, and subtle differences could confound the results. We mitigated this by using identical codebases with feature flags.

External Validity: Evaluation uses a DVD benchmark (synthetic) and one DeFi protocol (47 contracts). The results may not be generalized to all smart contract projects. The completeness assumption of the tool (A2 in the proof of

liveness) is strong, in practice, tools miss vulnerabilities, resulting in a probabilistic rather than a guaranteed liveness. The sample sizes (15 challenges, 20 audit sessions) are modest, and larger-scale evaluation is future work. Although the domain is smart contract auditing, the coordination patterns studied are not domain-specific and may apply to other long-running tool orchestration tasks, we do not claim generality beyond similar failure-prone workflows.

Construct Validity: F1-Score and TFCV are standard metrics but may not capture all dimensions of audit quality (e.g., report clarity, false positive cost). The PFIR success rate measures compilation/execution, not semantic correctness of the generated tests.

5.6 Reproducibility

Datasets: Damn Vulnerable DeFi benchmark (15 challenges), 500 security objectives from GitHub Solidity contracts, DeFi protocols (Uniswap V2, Compound V2, Aave V2). *Hyperparameters:* Risk weights $\alpha = 0.5$, $\beta = 0.3$, $\gamma = 0.2$, plan revision threshold $\theta = 0.15$, PFIR max attempts=5, *Sensitivity Analysis:* Risk weights ± 0.1 variation changes TFCV by $<8\%$, plan revision threshold $\theta \in [0.10, 0.20]$ balances responsiveness/stability, variation of confidence parameters ± 0.1 changes the repair success rate by $<3\%$ (robust). *LLM:* Claude Sonnet 4.5 via Anthropic API, average cost $2.30 per audit. *Environment:* Ubuntu 22.04, Python 3.10, Foundry 0.2.0, Slither 0.10.0, Mythril 0.24.0. Random seed: 42 for reproducibility.

6 Conclusion and Future Work

This paper presented SPEAR as an engineering case study of multi-agent coordination applied to autonomous smart contract auditing. The results suggest that explicit coordination protocols, local agent autonomy, and self-healing policies simplify recovery and adaptation in long-running, failure-prone workflows. This work illustrates how established coordination mechanisms can be composed to support robustness and resource-aware behavior in a realistic application domain.

Limitations: Tool completeness (A2) is a strong assumption, real tools miss vulnerabilities. The evaluation scale is modest. The Coordinator Agent (A_{Coord}) is a potential single point of failure.

Future work: Policy learning through RL, expanded tool integration, human-in-the-loop interfaces, and distributed coordination without central coordinator.

References

1. Alchourrón, C.E., Gärdenfors, P., Makinson, D.: On the logic of theory change: Partial meet contraction and revision functions. Journal of Symbolic Logic **50**(2), 510–530 (1985). https://doi.org/10.2307/2274239

2. Bellifemine, F., Poggi, A., Rimassa, G.: Developing multi-agent systems with jade. In: Castelfranchi, C., Lespérance, Y. (eds.) Intelligent Agents VII Agent Theories Architectures and Languages. pp. 89–103. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
3. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: Formal verification of smart contracts: Short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security. p. 91–96. PLAS '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2993600.2993611, https://doi.org/10.1145/2993600.2993611
4. Certora: Certora prover documentation. https://docs.certora.com/ (2026), accessed: April 6, 2026
5. Chen, X., Roşu, G.: —a semantic framework for programming languages and formal analysis. In: Engineering Trustworthy Software Systems: 5th International School, SETSS 2019, Chongqing, China, April 21–27, 2019, Tutorial Lectures. p. 122–158. Springer-Verlag, Berlin, Heidelberg (2019). https://doi.org/10.1007/978-3-030-55089-9_4, https://doi.org/10.1007/978-3-030-55089-9_4
6. Diligence, C.: Mythril - a symbolic-execution tool for evm bytecode. https://github.com/ConsenSysDiligence/mythril (Mar 2024), accessed: November 6, 2025
7. Durfee, E.H.: Distributed problem solving and planning. Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence pp. 121–164 (2001)
8. El Yamany, H.F., M. Capretz, M.A., Capretz, L.F.: A multi-agent framework for testing distributed systems. In: 30th Annual International Computer Software and Applications Conference (COMPSAC'06). vol. 2, pp. 151–156 (2006). https://doi.org/10.1109/COMPSAC.2006.98
9. Feist, J., Greico, G., Groce, A.: Slither: a static analysis framework for smart contracts. In: Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain. p. 8–15. WETSEB '19, IEEE Press (2019). https://doi.org/10.1109/WETSEB.2019.00008, https://doi.org/10.1109/WETSEB.2019.00008
10. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M.: Codebert: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020. pp. 1536–1547 (2020). https://doi.org/10.18653/v1/2020.findings-emnlp.139
11. Foundation for Intelligent Physical Agents: Fipa agent communication language specification. Standard SC00061J, FIPA (2000), http://www.fipa.org/specs/fipa00061/
12. Grieco, G., Song, W., Cygan, A., Feist, J., Groce, A.: Echidna: effective, usable, and fast fuzzing for smart contracts. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 557–560. ISSTA 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3395363.3404366, https://doi.org/10.1145/3395363.3404366
13. Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A., Rosu, G.: Kevm: A complete formal semantics of the ethereum virtual machine. In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF).

pp. 204–217. IEEE (2018). https://doi.org/10.1109/CSF.2018.00022, https://doi.org/10.1109/CSF.2018.00022
14. Ince, P., Yu, J., Liu, J.K., Du, X.: Generative large language model usage in smart contract vulnerability detection (2025), https://arxiv.org/abs/2504.04685
15. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003). https://doi.org/10.1109/MC.2003.1160055
16. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: A generic method for automatic software repair. IEEE Transactions on Software Engineering **38**(1), 54–72 (2012). https://doi.org/10.1109/TSE.2011.104
17. Long, F., Rinard, M.: Automatic patch generation by learning correct code. In: Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 298–312 (2016). https://doi.org/10.1145/2837614.2837617
18. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269. CCS '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2976749.2978309, https://doi.org/10.1145/2976749.2978309
19. Mehar, I., Shier, C., Giambattista, A., Gong, E., Fletcher, G., Sanayhie, R., Kim, H., Laskowski, M.: Understanding a revolutionary and flawed grand experiment in blockchain: The dao attack. Journal of Cases on Information Technology (JCIT) **21**, 19–32 (01 2019). https://doi.org/https://doi.org/10.4018/JCIT.2019010102
20. Rao, A.S., Georgeff, M.P.: Modeling rational agents within a bdi-architecture. In: Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning. p. 473–484. KR'91, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1991)
21. Rao, A.S., Georgeff, M.P.: Bdi agents: From theory to practice. Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95) pp. 312–319 (1995)
22. Sandholm, T.W.: Distributed rational decision making. Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence pp. 201–258 (1999)
23. Smith, R.G.: The contract net protocol: High-level communication and control in a distributed problem solver. IEEE Transactions on Computers **C-29**(12), 1104–1113 (1980). https://doi.org/10.1109/TC.1980.1675516
24. Sun, Y., Wu, D., Xue, Y., Liu, H., Wang, H., Xu, Z., Xie, X., Liu, Y.: 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. p. 1–13. ICSE '24, ACM (Apr 2024). https://doi.org/10.1145/3597503.3639117, http://dx.doi.org/10.1145/3597503.3639117
25. Wei, Z., Sun, J., Zhang, Z., Hou, Z., Zhao, Z.: Adaptive plan-execute framework for smart contract security auditing (2025), https://arxiv.org/abs/2505.15242
26. Werner, S., Perez, D., Gudgeon, L., Klages-Mundt, A., Harz, D., Knottenbelt, W.: Sok: Decentralized finance (defi). In: Proceedings of the 4th ACM Conference on Advances in Financial Technologies. p. 30–46. AFT '22, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3558535.3559780, https://doi.org/10.1145/3558535.3559780
27. Weyns, D.: An Introduction to Self-Adaptive Systems: A Contemporary Software Engineering Perspective. John Wiley & Sons (2020)
28. Woolridge, M., Wooldridge, M.J.: Introduction to Multiagent Systems. John Wiley & Sons, Inc., USA (2001)

29. Yu, X.L., Al-Bataineh, O., Lo, D., Roychoudhury, A.: Smart contract repair. ACM Trans. Softw. Eng. Methodol. **29**(4) (Sep 2020). https://doi.org/10.1145/3402450, https://doi.org/10.1145/3402450
30. Zaazaa, O., El Bakkali, H.: Smartllmsentry: A comprehensive llm based smart contract vulnerability detection framework. Journal of Metaverse **4**(2), 126–137 (2024). https://doi.org/10.57019/jmv.1489060