International Journal of Computer Science Engineering Techniques **(IJCSE)** Volume 10 Issue 3 ,May-June 2026

ISSN : 2455-135X

Koan: Safety-Validated Intent Compilation for DeFi Workflow Orchestration

Harshit Arora[1], Varun Singh[2], Abhinav Rajeev Kumar[3], Nanjappan Manikandan[4]

[1,2,3,4]Department of Data Science and Business Systems, SRM Institute of Science and Technology, Tamil Nadu, India.

1ha8052@srmist.edu.in

Received: Revised: Accepted: Published:

Abstract - *We introduce Koan, a system for compiling natural language DeFi requests into executable safety-validated directed acyclic graphs (DAGs). Assembling correct multi-step DeFi workflows requires sequencing irrevocable on-chain transactions across heterogeneous protocols, demanding flexible intent understanding and strict execution discipline simultaneously - a combination no existing tool provides. Koan addresses this in two phases. Phase 1 translates user intent into a typed graph via an LLM with deterministic fallback heuristics. Phase 2 validates that graph, injects missing safety nodes, and executes with dependency-aware scheduling. We evaluated on 1,000 prompts across 9 DeFi categories. Intent-to-workflow correctness reached 82.4%; DAG validity 93.6%. The Safety Injector raised price-impact check coverage from 41.2% to 98.4%, and 7.3% of all workflows were aborted by injected checks identifying excessive risk. Workflow authoring averaged 2.4 min versus 46.8 min for manual scripting (a 20x speedup), and compiled flows achieved 97% execution success with 18% gas savings on matched DEX routes under testnet conditions.*

Keywords - *Blockchain systems, decentralized finance, intent compilation, large language models, workflow orchestration.*

ISSN : 2455-135X

1. Introduction

A typical DeFi operation - routing ETH to WBTC at best price with a 1% impact cap and on-chain confirmation - requires at least six discrete components: routing APIs, token approval mechanisms, a price-impact oracle, slippage logic, a transaction broadcaster, and a status tracker. Missing any one step can cause irreversible loss [1]. These components expose heterogeneous APIs, so users must manually integrate them, validate inter-dependencies, and guard every edge case.

No existing tool closes this gap end-to-end. Workflow platforms (Zapier, n8n, Node-RED [2,3,4]) lack on-chain transaction ordering and gas awareness. LLM pipeline editors (Langflow, Flowise [5,6]) target retrieval-augmented generation, not stateful on-chain operations. DeFi-native aggregators (1inch, CoW [7,8]) optimize individual trades but provide no multi-step workflow abstraction.

We cast this as intent compilation: transforming a natural-language specification into a validated, executable workflow graph with intrinsic safety assurances. The fundamental tension is that intent capture suits probabilistic reasoning, but on-chain execution demands deterministic graph fidelity.

1.1. Contributions

This work makes four contributions:

- **Intent compiler framing.** We formalize NL-to-workflow translation as a typed compilation pipeline that segregates probabilistic intent inference from deterministic execution.
- **Hybrid architecture.** We build and evaluate a two-phase system combining an LLM mapper with deterministic fallback heuristics and a structural graph validator.
- **Safety-first orchestration.** Structural validity and risk-aware node injection are compiler invariants, yielding 96.2% unsafe-flow block rate and 98.4% price-impact coverage.
- **Benchmark.** We release 1,000 prompts across nine DeFi categories with three overlap subsets for fair cross-system comparison.

2. Background and Related Work

2.1. Workflow Automation and LLM Pipeline Builders

General-purpose workflow tools (Zapier [2], n8n [3], Node-RED [4], Make [9]) operate over HTTP webhooks and lack on-chain transaction ordering, gas awareness, or protocol-level parameter schemas. LLM pipeline editors Langflow [5] and Flowise [6] add drag-and-drop graph editing but target retrieval-augmented generation, not stateful on-chain operations. The broader "NL to typed intermediate to deterministic execution" pattern is established: Semantic Interpreter [10] compiles intent into Office DSL programs; Sqlizer [11] synthesizes SQL via sketch-and-repair. Koan applies the same pattern to DeFi node semantics with a constructive safety invariant and irrevocable on-chain execution. Agint [12] pursues a similar NL-to-typed-DAG philosophy for software engineering agents; Koan differs in targeting DeFi protocol semantics and enforcing domain-specific safety invariants by construction. WorfBench [13] documents that LLM-generated DAGs frequently break on dependency edges in longer chains, directly motivating Koan's explicit graph-validation pass.

2.2. DeFi Protocols, MEV, and Intent Systems

Werner et al. [14] identify composability risk as the central DeFi security challenge; interactions between independently safe protocols can produce unsafe aggregate behavior. MEV compounds this - Daian et al. [15] show frontrunning extracts hundreds of millions annually, with multi-transaction workflows especially exposed. CoW Protocol [8] and Fusion [7] resolve single-transaction intents via solver networks but do not support multi-step workflow abstraction. Know Your Intent [1] classifies DeFi intent into five categories via LLM but stops short of producing a validated executable graph; Koan extends to full DAG synthesis and safety enforcement. Cross-chain bridge choice is abstracted behind a unified node interface [16]; non-atomicity across chains remains open.

2.3. Smart Contract Safety and Visual Programming

Contract-level tools (Oyente [17], Securify [18]) verify individual contracts via symbolic execution; Koan operates one level above, enforcing safety by injection at the workflow layer - complementary approaches that could be combined. Kelleher and Pausch [19] identify node-based editors as a proven strategy for lowering programming barriers; Koan applies this to DeFi-specific semantics.

3. Materials and Methods

Koan has two phases. Phase 1 is the probabilistic half: discover intent, synthesize a typed DAG. Phase 2 is fully deterministic: validate structure, enforce safety, execute with dependency-aware scheduling. All uncertainty stays in Phase

ISSN : 2455-135X

1. By the time a graph reaches Phase 2, it is either valid or rejected.

3.1. Formal Workflow Model

A Koan workflow is a typed DAG G = (V, E, tau, kappa) where V is a finite set of protocol-operation nodes, E is a subset of V x V encoding data dependencies, tau maps each node to a type from the executor catalog, and kappa carries per-node configuration. G is structurally valid iff: (1) acyclicity - no directed path returns to its origin; (2) type compatibility - for each edge (u, v) in E, the output type of tau(u) matches the input type of tau(v); (3) schema completeness - kappa(v) provides all required fields for tau(v). For example, oneInchSwap mandates fromToken, toToken, chainId, and slippage; omitting slippage triggers immediate rejection.

G is safety-compliant if additionally: every swap node (tau(v) is oneInchSwap or fusionSwap) has an upstream priceImpactCalculator on some path, and every terminal transaction node has a downstream transactionMonitor. Given input x, the pipeline outputs G(x) satisfying both properties or halts with a diagnostic. These invariants are necessary but not sufficient for financial safety; full semantic equivalence to user intent is delegated to the human approval gate.

3.2. Implementation Architecture

Koan is a composition of three services. The Agent Service (implemented in Python/FastAPI) executes the Architecture Mapper and Workflow Generator. All subsequent stages are handled by the Backend Execution Service (TypeScript/Node.js), which includes the Graph Validator, Safety Injector, Execution Engine, and 15 node executors. The Frontend (React and React Flow) provides the visual canvas, configuration panels, and chatbot interface.

The Agent Service exposes POST /process (natural language to typed DAG), POST /approve-workflow (forward workflow to backend), and GET /executions/{id} (status polling). Conversation state is held in-memory per session. The Backend Service handles execution through POST /api/workflows/execute and exposes status and cancellation endpoints. Inter-service communication uses JSON/HTTP.

3.3. Phase 1: Intent Discovery and Workflow Synthesis

3.3.1. LLM Prompt Design and Fallback Heuristics

The system prompt instructs the LLM to return a JSON object with six fields: pattern (one of four workflow types), tokens, chains, features, suggestedNodes (ordered node type IDs), and confidence in [0, 1]. We include a full JSON schema and three few-shot examples (simple swap, cross-chain bridge, limit order) to lock down the output format.

If the input is not about DeFi, the LLM must return {"conversational": true}. We add three hardening layers: (1) schema-constrained output via OpenAI response_format: json_object; (2) node-type allowlisting (the generator drops any suggestedNodes entry not in the catalog); and (3) confidence gating - scores below 0.6 fall through to the deterministic heuristic. The fallback heuristic is a term-matching classifier over 43 domain terms in four bins: action words, protocol names, ERC-20 token symbols, and chain identifiers. The highest-scoring pattern above 2.0 wins.

3.3.2. Architecture Mapper

The Architecture Mapper takes raw NL input and returns a structured requirements object: workflow pattern, token/chain entities, required node types, and a confidence score. The LLM backend is pluggable; the prototype uses GPT-4o (OpenAI, temperature 0.0) with Anthropic Claude as an alternative. If the LLM call fails or confidence falls below threshold, control passes to the heuristic analyzer, which handles approximately 74% of simple swap requests with no LLM call at all.

3.3.3. Workflow Generator

The generator consumes the mapper's requirements and outputs a typed DAG: nodes with configuration schemas, directed edges for data dependencies, layout coordinates for the canvas, and workflow metadata (version, pattern, token context). Each recognized pattern has a pre-validated subgraph template behind it. The DEX aggregation template, for example, produces a 10-node chain: walletConnector -> tokenSelector -> oneInchQuote -> priceImpactCalculator -> oneInchSwap -> fusionSwap -> limitOrder -> portfolioAPI -> transactionMonitor -> defiDashboard. Cross-chain bridge patterns get different treatment: the generator swaps in fusionPlus and chainSelector nodes and rewires edges for cross-chain message ordering.

3.4. Phase 2: Validation and Deterministic Runtime

3.4.1. Graph Validator

No DAG reaches execution without passing four structural checks: (1) schema validation (all required config keys present); (2) dependency correctness (no dangling edge endpoints); (3) cycle detection via topological sort; (4) type

ISSN : 2455-135X

compatibility (output type of tau(u) must match input type of tau(v) for each edge). Failures are surfaced to the frontend with node-level highlighting and field diagnostics.

3.4.2. Safety Injector

After structural validation, the injector scans for missing risk controls: a swap node (oneInchSwap or fusionSwap) without an upstream priceImpactCalculator gets one injected; on-chain terminal nodes without a downstream transactionMonitor get one appended. All injections are deterministic and logged; injected nodes are visually marked on the canvas. One caveat: priceImpactCalculator queries a pre-execution spot-price quote from the 1inch Swap API - a best-effort check, not an on-chain simulation. Liquidity shifts between quote and execution can still cause slippage to exceed the estimate.

3.4.3. Human Approval Gate

No on-chain action fires until the user says so. The approval screen shows the full graph, node configurations, which safety nodes were injected, estimated gas, and a risk summary. This is a hard gate, not a soft warning. ERC-4337 smart-contract wallets could enforce per-workflow spending policies at the wallet layer itself, providing a cryptographic complement to Koan's workflow-level approval gate.

3.4.4. Execution Engine

After approval, the engine topologically sorts the DAG and runs independent nodes in parallel, passing each node's output into a shared execution context map. Workflow state is one of {pending, running, completed, failed, cancelled}; node failures emit a structured payload with reason, step, and recovery options.

3.5. Node Executor Catalog

Table 1 lists the 15 node types registered in the executor catalog. Each node type has a corresponding executor class implementing three methods: validate(config) for precondition checks, execute(context) for the actual protocol operation, and getTemplate() returning the configuration skeleton the Workflow Generator uses when hydrating node defaults.

Table 1. Registered Node Executor Catalog

Node Type	Category	Operation
walletConnector	Infra	Wallet auth / signer
tokenSelector	Infra	Token address resolution
chainSelector	Infra	Chain RPC config
erc20Token	Infra	ERC-20 metadata
oneInchQuote	DEX	1inch quote fetch
oneInchSwap	DEX	1inch swap execution
fusionSwap	DEX	Fusion order routing
fusionPlus	X-chain	Fusion+ bridge
fusionMonadBridge	X-chain	Monad bridge relay
limitOrder	Strategy	Limit order placement
priceImpactCalculator	Safety	Slippage estimation
transactionMonitor	Safety	On-chain status
transactionStatus	Safety	Receipt polling
portfolioAPI	Analytics	Balance aggregation
defiDashboard	Analytics	Performance dashboard

4. Results and Discussion

4.1. Evaluation Setup

We built a benchmark of 1,000 NL DeFi prompts across nine categories: simple swaps (120), cross-chain (120), limit orders (120), multi-step portfolio (120), yield farming (80), liquidation protection (80), governance (80), hybrid/novel (80), and deliberately ambiguous (200). Two annotators built ground-truth graphs for all non-ambiguous categories (Cohen's kappa = 0.83). Every prompt runs 5 times. LLM calls use GPT-4o at temperature 0.0. Execution tests run on Ethereum Sepolia and a Hardhat mainnet fork. All latency and gas numbers are testnet. We compare eight systems on a 480-task non-ambiguous subset with three overlap subsets (80/120/160 tasks) that include only tasks every system can attempt.

4.2. Intent Mapping Accuracy

How reliably does the hybrid mapper identify patterns and build valid graphs as prompts grow more complex? Table 2 breaks down results by category. Edge F1 runs 2-4 points below Node F1 across the board (wiring edges is harder than selecting nodes), and strongly predicts execution failures: categories with the lowest edge scores (hybrid: 0.58, multi-step portfolio: 0.72) have the highest failure rates (9.5%, 6.8%), Pearson r = -0.92. The graph validator is not a formality. The four newer categories (yield farming, liquidation protection, governance, hybrid) land at 65-74% accuracy due to sparse template coverage; the ambiguous category (58.5%) characterizes deliberate underspecification.

Table 2. Intent Mapping Accuracy by Prompt Category

Category	N	Pat. Acc.	Node F1	Edge F1	DAG Val.	Exec. Fail
Simple Swap	120	95.0%	0.91	0.89	98.1%	2.8%
Cross-Chain	120	92.0%	0.88	0.85	95.2%	4.5%
Limit Orders	120	88.0%	0.84	0.81	92.5%	5.2%
Multi-Step	120	78.0%	0.76	0.72	87.8%	6.8%

ISSN : 2455-135X

Portfolio						
Yield Farming	80	73.5%	0.71	0.68	83.8%	8.2%
Liquidation Prot.	80	72.5%	0.70	0.66	82.5%	-
Governance	80	71.0%	0.68	0.64	81.2%	-
Hybrid / Novel	80	65.0%	0.62	0.58	73.8%	9.5%
Ambiguous	200	58.5%	0.51	0.47	61.2%	-
Overall	1000	82.4%	0.77	0.73	93.6%	-

4.3. Safety Impact

Does the deterministic injector improve safety coverage over what the LLM includes on its own? Table 3 tells a clear story. With the injector off, only 41.2% of swap workflows include a price impact check. With it on: 98.4%. The false injection rate (4.2%) is how often the injector adds nodes to workflows that were already safe; low enough that usability is not hurt. The abort rate (7.3%) is how often the injected priceImpactCalculator actually fires at execution time, halting the workflow before a risky swap goes through. Safety nodes are not just present; they actively block trades in a non-trivial fraction of runs.

Table 3. Safety Injector Effectiveness

Safety Metric	Koan	No Injector
Unsafe-flow block rate	96.2%	0%
Risk-node injection recall	93.7%	-
Price impact check coverage	98.4%	41.2%
Tx monitoring coverage	97.1%	55.8%
False injection rate	4.2%	-
Impact threshold exceeded (abort)	7.3%	-

4.4. Operational Performance

What do compiled workflows cost in latency and gas, and how reliable are they? All figures in Table 4 are testnet (Sepolia + Hardhat fork). Simple swaps finish at 2.1 s p50 with 97.2% success and 18% gas savings on equivalent routes. The 18.2 s cross-chain latency is dominated by Fusion+ bridge round-trip, not orchestration overhead (avg. 340 ms). Multi-step portfolios consume the most gas (312,500 units) due to multi-contract interactions.

Table 4. Execution Performance by Flow Type (Testnet)

Category	p50 (s)	p95 (s)	Success	Gas
Simple Swap	2.1	3.8	97.2%	175,200
Limit Orders	3.2	5.4	94.8%	198,400
Multi-Step Portfolio	8.5	14.2	93.2%	312,500
Cross-Chain	18.2	30.8	94.2%	242,100
Manual Baseline	3.1	5.8	98.0%	219,543

4.5. Comparative Positioning

How does Koan compare to general automation platforms and DeFi-native tools on matched tasks? On the full 480-task set, Koan (90.2%) leads Langflow (46.5%), Flowise (44.8%), and n8n (39.8%) by wide margins, largely because DeFi-native tasks are out of reach for those platforms. The more informative comparison is the overlap subsets. On the 80-task minimal overlap, Koan (85.0%) still leads by 16-21 percentage points, confirming the gains persist when task space is normalized. Hardhat Tasks (99.8%, 18.5 min) is the expert-practitioner ceiling; Koan is 7.7x faster (2.4 vs. 18.5 min), and 20x faster than raw manual scripting.

Table 5. Quantitative Baseline Comparison

System	Full 480	Overlap-80	Time (min)	Error	DeFi
Koan	90.2%	85.0%	2.4	3.2%	Full
Langflow	46.5%	68.8%	18.6	21.5%	None
Flowise	44.8%	66.2%	17.2	24.2%	None
n8n	39.8%	63.8%	21.8	28.8%	Partial
Node-RED	37.2%	-	24.2	30.2%	Partial
Make	35.2%	-	22.5	31.8%	None
Hardhat Tasks	99.8%	-	18.5	4.2%	Full
Manual Scripting	100%	100%	46.8	8.5%	Full

4.6. Ablation Study

Table 6 isolates each component. Removing the graph validator drops DAG validity from 94.3% to 67.8%, the largest single degradation - the LLM produces structurally broken graphs far more often than intuition suggests. Removing fallback heuristics costs 3.5 pp in pattern accuracy, concentrated in simple swaps. Removing both yields the worst configuration on every metric, confirming the two mechanisms are complementary.

Table 6. Ablation Results (Overall Accuracy / DAG Validity Rate)

Configuration	Pattern Acc.	DAG Validity
Full Koan	82.6%	94.3%
No fallback heuristics	79.1%	94.0%
No graph validator	82.3%	67.8%
No safety injector	82.5%	93.9%
No fallback + no validator	75.4%	63.2%

4.7. Template-Matched vs. Novel Patterns

Templates do the heavy lifting: for recognized patterns the LLM only classifies and fills slots (91.2% accuracy); for novel or hybrid prompts outside the library the LLM must synthesize the full graph and accuracy drops 25.6 pp to 65.6%. The most direct improvement is adding templates for yield farming, liquidation protection, and governance.

Table 7. Template-Matched vs. Novel Pattern Accuracy

Mode	N	Pattern Acc.	Node F1	Edge F1
Template-	480	91.2%	0.88	0.85

ISSN : 2455-135X

matched				
Novel / hybrid	160	65.6%	0.62	0.58
Gap	-	25.6 pp	0.26	0.27

4.8. Adversarial Robustness

We ran four adversarial scenarios on a Hardhat mainnet fork. Under 3x price volatility the abort rate rises from 7.3% (nominal) to 22%, showing the safety check scales with the threat. Fusion MEV protection gives 91% success under sandwich simulations vs. 72% unprotected (19 pp). Oracle anomaly detection reaches 87.5% at 3.2% FPR. Mempool congestion degrades success to 76% at 10x gas, but those are timeouts, not logic failures. Under moderate adversarial pressure the safety model holds.

4.9. Discussion

The compilation framing holds up in practice. Typed intermediates, structural validation, and explicit execution semantics make workflows more reliable and accessible than manual scripting; the two-phase split also allows independent evolution of the LLM backend, node catalog, and validation rules. The clearest accuracy gap is the ambiguous category (58.5%, 0.47 edge F1), which requires a follow-up clarification turn. Making safety a compiler invariant gives three properties probabilistic generation cannot: completeness, auditability, and composability. The 7.3% abort rate confirms injected checks actively stop trades.

4.10. Limitations

All latency and gas figures are testnet; mainnet MEV and congestion will shift them. Accuracy is pinned to one GPT-4o snapshot; the 74% fallback floor is provider-agnostic but low. Edge F1 conservatively penalizes topologically equivalent orderings; a graph-isomorphism-aware metric would be fairer. Session and execution state are single-threaded; multi-user deployment requires concurrent-safe management. No formal user study has yet been conducted.

5. Conclusion

Intent compilation is a workable systems abstraction for DeFi automation. Splitting probabilistic intent inference from deterministic validation and execution resolves a real tension: LLMs are flexible but unreliable, and on-chain calls are unforgiving. Three results stand out. First, the graph validator is the single largest correctness contributor: DAG validity drops from 94.3% to 67.8% without it, confirming that structural enforcement - not LLM quality - is the primary reliability lever. Second, the Safety Injector reaches 96.2% unsafe-flow block rate independent of LLM output quality, and injected checks actively abort 7.3% of trades at execution time, demonstrating that safety-by-construction at the workflow layer provides guarantees probabilistic generation alone cannot. Third, Koan reduces workflow construction time 20x versus manual scripting while maintaining 97% execution success and 18% gas savings on matched routes under testnet conditions. The path forward is interactive clarification for ambiguous prompts, control-flow extensions beyond pure DAGs, ERC-4337 integration for wallet-layer policy enforcement, and a formal user study with non-technical DeFi participants.

Conflicts of Interest

The authors declare that there is no conflict of interest regarding the publication of this paper.

Funding Statement

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

Acknowledgments

The authors thank the Department of Data Science and Business Systems, SRM Institute of Science and Technology, for institutional support, and Prof. Nanjappan Manikandan for guidance throughout the project.

References

[1] Q. Mao, Y. Zhang, J. Chen, W. Zhou, and J. Yan, "Know Your Intent: An Autonomous Multi-Perspective LLM Agent Framework for DeFi User Transaction Intent Mining," arXiv preprint arXiv:2511.15456, 2025. [CrossRef] [Google Scholar] [Publisher Link]

[2] Zapier, "Zapier Apps Directory," 2026. [Online]. Available: https://zapier.com/apps. [Publisher Link]

[3] n8n, "n8n: AI Workflow Automation Platform," 2026. [Online]. Available: https://n8n.io/. [Publisher Link]

[4] Node-RED, "Node-RED: Low-Code Programming for Event-Driven Applications," 2026. [Online]. Available: https://nodered.org/. [Publisher Link]

[5] Langflow, "Langflow Documentation," 2026. [Online]. Available: https://docs.langflow.org/. [Publisher Link]

[6] Flowise, "Flowise Documentation," 2026. [Online]. Available: https://docs.flowiseai.com/. [Publisher Link]

[7] 1inch Network, "1inch Swap API," 2026. [Online]. Available: https://1inch.dev/swap-api. [Publisher Link]

[8] CoW Protocol, "CoW Protocol Documentation: MEV Protection," 2026. [Online]. Available:

ISSN : 2455-135X

https://docs.cow.fi/cow-protocol/concepts/benefits/mev-protection. [Publisher Link]

[9] Make, "Make: Visual-First Automation and AI Platform," 2026. [Online]. Available: https://www.make.com/en. [Publisher Link]

[10] A. Gandhi, T. Q. Nguyen, H. Jiao, R. Steen, and A. Bhatawdekar, "Natural Language Commanding via Program Synthesis," arXiv preprint arXiv:2306.03460, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[11] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig, "SQLizer: Query Synthesis from Natural Language," Proceedings of the ACM on Programming Languages, vol. 1, no. OOPSLA, pp. 63:1-63:26, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[12] A. Chivukula, J. Somasundaram, and V. Somasundaram, "Agint: Agentic Graph Compilation for Software Engineering Agents," arXiv preprint arXiv:2511.19635, 2025. [CrossRef] [Google Scholar] [Publisher Link]

[13] S. Qiao, R. Fang, Z. Qiu, X. Wang, N. Zhang, Y. Jiang, P. Xie, F. Huang, and H. Chen, "Benchmarking Agentic Workflow Generation," arXiv preprint arXiv:2410.07869, 2024. [CrossRef] [Google Scholar] [Publisher Link]

[14] S. M. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. J. Knottenbelt, "SoK: Decentralized Finance (DeFi)," Proceedings of the 4th ACM Conference on Advances in Financial Technologies (AFT '22), pp. 30-46, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[15] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability," Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP '20), pp. 910-927, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[16] A. Zamyatin, M. Al-Bassam, D. Zindros, E. Kokoris-Kogias, P. Moreno-Sanchez, A. Kiayias, and W. J. Knottenbelt, "SoK: Communication Across Distributed Ledgers," Proceedings of the 25th International Conference on Financial Cryptography and Data Security (FC '21), pp. 3-36, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[17] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16), pp. 254-269, 2016. [CrossRef] [Google Scholar] [Publisher Link]

[18] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bunzli, and M. Vechev, "Securify: Practical Security Analysis of Smart Contracts," Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), pp. 67-82, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[19] C. Kelleher and R. Pausch, "Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers," ACM Computing Surveys, vol. 37, no. 2, pp. 83-137, 2005. [CrossRef] [Google Scholar] [Publisher Link]