Generative Models for Writing Solana Smart Contracts: Comparative Analysis of Performance and Code Metrics

Emanuele Cuncu, Andrea Pinna, Giacomo Francesco Ibba, Gavina Baralla, Gianni Fenu, Roberto Tonelli
Department of Mathematics and Computer Science
University of Cagliari
Cagliari, Italy
Email: e.cuncu@studenti.unica.it, pinna.andrea@unica.it, giacomof.ibba@unica.it,
gavina.baralla@unica.it, fenu@unica.it, roberto.tonelli@unica.it

***Abstract*—Large Language Models (LLMs) have shown promising capabilities in code generation, yet their application to blockchain smart contracts for high-performance platforms like Solana remains largely unexplored. This paper presents a controlled experimental study evaluating four state-of-the-art LLMs, Claude Sonnet 4.5, GPT-5, Copilot, and DeepSeek V3.2, in generating Solana smart contracts using the Anchor framework. Our research addresses three main objectives: first, assessing the ability of LLMs to generate correct and functional Anchor code; second, exploring prompt engineering techniques to identify successful strategies and common failures; third, classifying hallucinations into syntactic, logical, and interpretive errors using quantitative code metrics. We establish a benchmark of 15 use cases with automated test generation and systematic prompt selection. Results show significant differences across models and prompt strategies, with structured approaches reducing critical errors. Our findings provide practical insights into LLM capabilities and limitations for smart contract generation, enabling better understanding of error patterns and establishing procedures for their detection. This contributes to addressing key challenges in AI-assisted blockchain development within the Solana ecosystem.**

***Index Terms*—LLM, Smart Contracts, Solana, Prompt Engineering, Hallucinations, Code Metrics**

I. Introduction

Large Language Models (LLMs) represent a substantial shift in artificial intelligence, demonstrating capabilities in code generation that extend beyond traditional programming paradigms. Within the blockchain domain, smart contract development presents unique challenges where correctness is paramount: errors can lead to irreversible economic losses and security vulnerabilities due to the immutable nature of distributed ledgers. While research has explored LLM-assisted contract generation for established platforms like Ethereum, high-performance ecosystems such as Solana remain largely unexamined. Solana distinguishes itself through architectural innovations achieving throughput significantly exceeding traditional blockchains. Smart contracts on Solana, termed "programs", execute within the Berkeley Packet Filter Virtual Machine and are typically developed in Rust using the Anchor framework. However, the explicit account management model and complex Program Derived Address handling, introduce substantial development complexity compared to account-abstracted platforms.

The application of LLMs to this domain raises critical questions. Current models exhibit tendencies towards hallucinations, which are plausible yet incorrect outputs that manifest as syntactic errors, logical inconsistencies, or misuse of deprecated APIs. In smart contract generation, such errors can compromise functional correctness even when code compiles successfully. Existing literature on code generation hallucinations has primarily focused on general-purpose programming or Solidity contracts, leaving a gap in understanding how these phenomena manifest in Solana's distinct programming model. This paper addresses three objectives: evaluating LLM capability to generate correct Anchor code, exploring prompt engineering strategies to identify effective approaches and common failure modes, and classifying hallucinations through quantitative code metrics. We establish a benchmark of 15 use cases with automated test generation, examining four models (Claude Sonnet 4.5, GPT-5, Copilot, DeepSeek V3.2) across three selected prompt strategies. Results reveal substantial performance variation, with structured prompt reducing critical errors while exposing model-specific limitations in handling complex contract logic.

This work was partially supported by the project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan, funded by the European Union–NextGenerationEU. Additionally, this research is part of the "IMASS CHAIN - Infrastructure Management Support System Chain," co-funded under the National Military Research Plan 2020, with CIG: 884399685F and CUP: D84H22001380001.

II. Related works

A. LLMs for code and smart contract generation

Transformer-based language models have demonstrated capability in generating syntactically valid code across multiple programming languages. Models such as Codex showed that pre-training on code repositories enables few-shot learning of programming patterns without task-specific fine-tuning. Subsequent developments have focused on alignment through

reinforcement learning from human feedback and specialisation through continued pre-training on code corpora. Research on prompt engineering has identified techniques that influence generation quality. Brwon et al. [1] demonstrated few-shot prompt where example pairs guide model behaviour without retraining. Wei et al. [2] showed that chain-of-thought prompt, requiring explicit intermediate reasoning steps, improves performance on logic-intensive tasks. Khojah et al. [3] systematically evaluated prompt strategies for function generation, finding that function signatures and package specifications reduce API-related errors, while persona and chain-of-though techniques improve code clarity although with marginal impact on correctness.

B. Hallucinations in code generation

Hallucinations in LLM outputs occur when models produce plausible yet factually incorrect content. In natural language, these manifest as made-up facts or logical contradictions. For code generation, Liu et al. [4] established a taxonomy distinguishing intent-conflicting hallucinations (deviation from requirements), context-deviation hallucinations (internal inconsistencies, code repetition, unreachable instructions), and knowledge-conflicting hallucinations (incorrect API usage or deprecated functions). Their analysis of over 3,000 code samples revealed that intent conflicts and inconsistencies comprise approximately 64% of hallucinations, whilst knowledge conflicts account for 15%. Laneve et al. [5] show that LLMs often make mistakes when comparing programs that should behave the same, even when the differences come only from basic compiler-style changes. Literature on smart contract generation has concentrated on Solidity. Petrović et al. [6] demonstrated that models can generate compilable contracts but identified frequent logical errors requiring manual intervention. Napoli et al. [7] observed similar patterns, noting that while syntactic correctness is often achieved, functional correctness remains inconsistent. Baralla et al. [8], [9] in their work evaluated Github's Copilot and Deepseek-V3 and Deepthink-R1 highlighting strengths and weaknesses of LLMs for smart contract generation, contract repair, and vulnerability detection. No systematic study has examined LLM performance for Solana's Anchor framework, where the account-centric programming model and explicit state management present distinct challenges from Ethereum's account-abstracted approach.

III. Methodology

This section presents our experimental framework for systematic evaluation of LLM performance in generating Solana smart contracts. Fig. 1 illustrates our controlled pipeline, encompassing model selection, use case definition, prompt strategy screening, code generation, and comprehensive evaluation through metrics and hallucination classification. The subsequent subsections detail each component of this methodology.

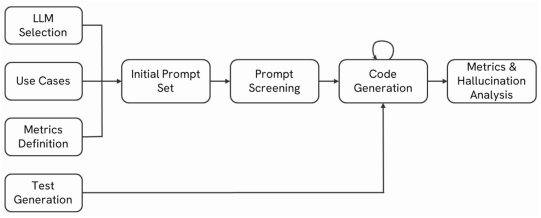

Fig. 1. Experimental Pipeline Workflow

A. Experimental Pipeline

To ensure reproducibility and consistency across all experiments, we designed a structured pipeline that controls the generation and evaluation process while minimizing the inherent variability of LLMs, as represented in Figure 1. **Phase 1: Initial Setup**:*(i) Model selection* Four state-of-the-art LLMs representing proprietary and open-source approaches (Claude Sonnet 4.5, GPT-5, Copilot, DeepSeek V3.2); *(ii) Benchmark definition*: 15 representative smart contract use cases selected from established Solana repositories [10]; *(iii) Evaluation metrics*: a set of quantitative measures including code structure, complexity, and correctness indicators.
Phase 2: Prompt Strategy Selection. We created an initial set of nine prompt variants combining different engineering techniques (persona, chain-of-thought, few-shot, function signature, list-of-packages). Using a pilot use case we generated contracts with all strategies and evaluated their performance based on iterations required, code metrics, and hallucination occurrence. This screening phase identified the three most effective strategies based on the lowest hallucination rate and the fewest required iterations: PS2 (function signature + persona), PS7 (function signature + chain-of-thought + list-of-packages), and PS9 (combination of all techniques).
Phase 3: Code Generation. For each of the 15 use cases, we generated smart contracts using the three selected prompt strategies across all four models, resulting in 180 programs. Each generation followed an iterative refinement process where compilation errors and test failures were feedback to the model, with a maximum limit of 15 iterations per contract.
Phase 4: Evaluation and Analysis. For each generated program, we measured code quality metrics, tracked the number of iterations required for compilation and functional correctness, and manually classified hallucinations according to our taxonomy. All experimental artifacts, including prompts, generated code, test suites, and analysis scripts, are publicly available[1].

B. Evaluation Metrics

We evaluated generated programs using three categories of metrics: generation efficiency, code structure quality, and hallucination classification. **Generation Metrics**. We measured two key efficiency indicators: *(i)* iterations required to obtain compilable code, and *(ii)* iterations needed for functionally correct code that passes all tests. We established a maximum

[1]Artifacts on OSF: https://shorturl.at/fEW2V

threshold of 15 iterations per contract. After the initial generation, models received only compilation errors or test failure messages as feedback. Additional textual instructions were provided only when models entered repetitive cycles without progress. **Code Structure Metrics**. We analyzed the following quantitative measures: *(i) Lines of Code (LoC)*: Program length; *ii) Comment Density (CpL)*: Ratio of comments to code lines, indicating maintainability; *iii) Number of Declared Functions (NDF)*: Measure of modularity; *iv) Lines per Function (LpF)*: Average function length, providing insight into code complexity; *v) McCabe Cyclomatic Complexity (MCC)*: Number of independent logical paths computed with the Lizard library. *vi) CrystalBleu Similarity* [11]: A code-specific similarity metric based on BLEU. We calculated similarity both between generated code and reference implementations, and among programs generated by the same model with different prompts to identify consistent generation patterns.

C. Use Cases

We selected 15 smart contract use cases from the benchmark established by Bartoletti et al. [10], choosing contracts with complete Anchor implementations from their publicly available repository. These use cases represent a diverse set of common decentralized applications, ranging from simple transfers to complex DeFi protocols, providing a robust foundation for evaluation.Table I lists the selected use cases along with their reference implementation metrics. The benchmark includes fundamental operations (transfers, storage), financial primitives (escrow, auction, crowdfunding), security mechanisms (vault, vesting), betting systems with oracles, and advanced DeFi components (AMM). All contracts were verified and updated to Anchor version 0.31.1, with minor modifications to two contracts (Escrow and Vault) limited to account serialization without altering their core logic. The reference implementations exhibit significant structural variability. Lines of code range from 92 to 478, cyclomatic complexity from 1 to 10, and the number of functions from 2 to 11. This diversity enables evaluation across different complexity levels, from straightforward contracts to sophisticated multi-function systems, providing insights into how contract complexity affects LLM generation performance.

TABLE I
BENCHMARK USE CASES AND REFERENCE METRICS

ID	Use Case	LoC	CpL	NDF	LpF	MCC
UC1	Bet	175	0.02	3	21.0	6
UC2	Simple Transfer	109	0.01	2	25.0	7
UC3	Token Transfer	186	0.06	2	45.0	10
UC4	HTLC	138	0.04	3	18.7	5
UC5	Escrow	206	0.03	4	21.3	6
UC6	Auction	183	0.04	3	28.0	7
UC7	Crowdfund	217	0.04	4	27.5	9
UC8	Vault	199	0.05	4	21.0	4
UC9	Vesting	173	0.03	5	19.4	10
UC10	Storage	92	0.02	3	4.7	1
UC11	Simple Wallet	214	0.00	4	21.3	3
UC12	Price Bet	225	0.03	4	26.0	5
UC13	Payment Splitter	199	0.01	8	14.3	8
UC14	Lottery	316	0.01	11	15.5	7
UC15	AMM	478	0.04	4	75.5	9

D. LLM Model Selection

We evaluated four state-of-the-art LLMs representing different approaches to code generation: Claude Sonnet 4.5, GPT-5, DeepSeek V3.2, and Copilot. All models were accessed through their free web interfaces to ensure reproducibility. In this setup, parameters such as temperature, top-p, and maximum output length were determined by each platform's defaults.

The selection provides diverse coverage: Claude Sonnet 4.5 (top performer in coding benchmarks [12]), GPT-5 (general-purpose with extended reasoning capabilities), DeepSeek V3.2 (open-source, code-focused), and Copilot (GPT-based, optimized for development environments). This combination enables comparison across high-performance generalists, specialized coding tools, and open-source alternatives.

E. Hallucination Classification

We adopted a structured classification framework to identify and categorize errors in generated code, based on the taxonomy proposed by Liu et al. [4]. We classified hallucinations into five categories: *H1 - Intent Conflicting*: Generated code deviates from the specified requirements or prompt instructions. *H2 - Inconsistency*: Logical contradictions within the code, such as mismatched variable names or incompatible conditions. *H3 - Repetition*: Redundant code blocks or duplicated logic. *H4 - Dead Code*: Unreachable instructions or unused variable declarations. *H5 - Knowledge Conflicting*: Incorrect API usage, including deprecated modules or non-existent functions.

Detection Methodology. We combined automated and manual analysis. Dead Code and Knowledge Conflicting hallucinations were primarily detected automatically through compilation warnings and error messages. Intent Conflicting, Inconsistency, and Repetition required manual inspection through source code review and test execution analysis. More specifically, Repetition hallucinations were identified by examining the source code for duplicate lines or portions of code. Intent Conflicting and Inconsistency were initially identified by observing failed tests or failed compilations, since by their nature, the presence of these hallucinations causes tangible problems with the correct functioning of the programs. For each generated contract, we created a detailed report documenting all identified hallucinations with code excerpts and explanations.

F. Prompt Strategy Selection

Initially, we tested nine variants combining different prompt engineering techniques (Table II), all including Function Signature as base and essential PDA/signer specifications for test compatibility. We selected three prompt strategies through systematic evaluation on a pilot use case based on the lowest hallucination rate shown in Table II and the fewest required iterations: *PS2 (Function Signature + Persona)*: Establishes expert context and coding standards. Second-best iteration performance with low hallucination rate. *PS7 (Function Signature + Chain-of-Thought + List-of-Packages)*: Guides step-by-step

TABLE II
PROMPT STRATEGIES AND HALLUCINATION DISTRIBUTION

ID	Strategy	H1	H2	H3	H4	H5	Clean
PS1	Function signature	0	1	0	1	0	2
PS2	+ Persona	0	1	0	0	1	3
PS3	+ Chain-of-thought	0	1	0	0	0	3
PS4	+ List-of-packages	0	2	0	1	0	2
PS5	+ Few-shot	0	0	0	0	1	3
PS6	+ Persona + List	0	0	0	1	0	3
PS7	+ CoT + List	0	0	0	1	0	3
PS8	+ Few-shot + List	0	0	0	1	0	3
PS9	All combined	0	0	0	1	0	3

TABLE III
TEST GENERATION ITERATIONS AND TEST COUNT PER USE CASE

UC	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Iter.	5	3	6	10	10	5	3	3	5	9	4	7	5	14	12
Tests	17	16	10	19	19	17	21	23	12	11	20	16	17	17	18

reasoning while constraining API usage to available functions. *PS9 (All Techniques Combined)*: Best performance, combining persona, chain-of-thought, few-shot examples, function signatures, and package specification.

G. Test Generation

To validate functional correctness, we generated comprehensive TypeScript test suites for each use case using Claude Sonnet 4.5. We initially attempted to use all four models for test generation, but only Claude consistently produced executable and correct test code. Tests from other models failed to interact properly with contracts even after multiple iterations. For this reason, we relied exclusively on Claude. This choice may introduce a model-specific bias in test structure and difficulty. Our test generation approach provides the model with the reference implementation and a structured prompt specifying: (i) function signatures to test, (ii) requirements for generic and reusable test design, and (iii) implementation guidelines including dynamic error handling, lamport verification, PDA derivation, and behavioral focus rather than implementation-specific checks. Table III shows iterations and test counts per use case. Iteration requirements ranged from 3 to 14, primarily depending on contract complexity. More complex cases (UC14, UC15) required more iterations to achieve fully executable test suites. The number of generated tests remained relatively uniform across use cases (10-23 tests), with balanced coverage across contract functions, indicating comprehensive testing without excessive focus on individual components.

IV. RESULTS

A. Generation Performances

The generation process resulted in an output set of 180 programs. Table IV shows the number of iterations required for each use case and every combination of model and prompt strategy to *(i)* obtain compiling code and *(ii)* obtain a test-compliant code. Notably, only UC12, generated by the combination of Copilot-PS2, resulted in no compiling code within the 15 iterations. In the other cases, the average number of

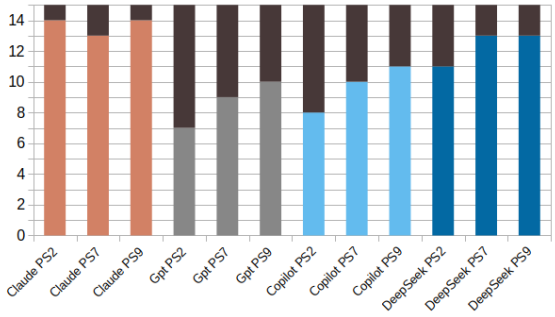

Fig. 2. Number of successful generations per model and prompt strategy

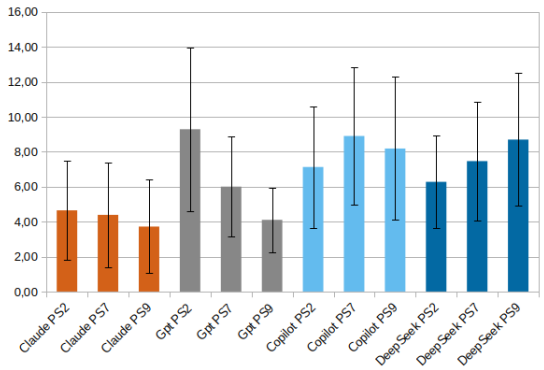

Fig. 3. Average number of iterations and confidence interval before generating a test-compliant code per model and prompt strategy

iterations to achieve compiling code is $3.29 \pm 23.9\%$. Among these, the most efficient model-prompt strategy is Claude-PS9, which averages $2.00 \pm 68.1\%$ iterations. When evaluating the total number of iterations required to obtain code that successfully passes the tests, we observed numerous failures to stay within the limit of 15 iterations. Notably, for UC15 (AMM) no combinations produced a test-compliant program. Conversely, all combinations successfully implemented UC1. Overall, UC7 required the minimum number of iterations both to get a compiling code (on average 1.8 iterations) and for having the program passes the test (on average 3.5 iterations).

As illustrated in Fig. 2, the number of use cases for which the models produce successful programs varies significantly according to the prompt strategy. Notably, PS9 consistently results in a higher number of successful programs compared to prompt strategy 2. Only with Claude did the PS7 strategy yield fewer successful programs than PS2. In total, 47 out of 180 programs failed to pass the tests after 15 iterations. The most successful model is Claude, which generated 41 of 45 test-compliant use cases within 15 iterations. The most effective combination is Claude-PS9, with 14 out of 15 successful programs and an average of $3.71 \pm 72\%$ iterations (computed without considering UC15). For comparison, Fig. 3 illustrates the average number of iterations for each combination, considering only the successful cases resulting in test-compliant programs.

TABLE IV

NUMBER OF ITERATIONS TO OBTAIN THE COMPILATION AND THE TEST-COMPLIANCE PER MODEL-PROMPT STRATEGY-USE CASE

FORMAT: THE FIRST NUMBER IS THE NUMBER OF ITERATIONS TO OBTAIN A COMPILING CODE. THE SECOND NUMBER IS THE TOTAL NUMBER OF ITERATIONS TO OBTAIN A PROGRAM THAT PASSES THE TEST. F = FAILURE WITHIN 15 ITERATIONS

Model-PS	**UC1**	**UC2**	**UC3**	**UC4**	**UC5**	**UC6**	**UC7**	**UC8**	**UC9**	**UC10**	**UC11**	**UC12**	**UC13**	**UC14**	**UC15**
Claude PS2	4, 4	2, 2	3, 4	1, 6	5, 8	1, 1	2, 3	4, 5	4, 12	1, 2	1, 5	1, 6	2, 4	1, 3	1, F
Claude PS7	1, 1	2, 2	2, 6	1, 6	4, 5	1, 1	1, 3	2, 3	4, 10	1, 2	5, 7	5, 9	2, F	2, 2	2, F
Claude PS9	1, 1	2, 2	4, 7	2, 10	6, 7	1, 3	1, 1	1, 1	2, 4	1, 3	2, 3	2, 5	2, 3	2, 2	1, F
GPT PS2	3, 14	5, F	5, F	3, F	4, F	2, F	1, F	5, 11	5, F	1, 2	1, 12	8, 14	4, 7	4, 5	4, F
GPT PS7	5, 9	5, 6	11, 11	3, F	3, F	6, F	2, 3	3, 7	5, 5	5, F	2, 7	2, 2	4, 4	9, F	6, F
GPT PS9	3, 4	5, 6	5, 5	1, 4	6, F	3, 4	2, 2	3, 3	6, F	1, 2	3, 8	2, 3	4, F	4, F	3, F
Copilot PS2	3, 4	3, 8	4, 9	5, F	4, F	2, 13	2, 3	4, 10	2, F	3, F	4, F	F, F	4, 6	3, 4	4, F
Copilot PS7	3, 9	4, 11	4, 4	5, F	5, 14	4, 15	2, 3	6, 10	6, F	5, F	3, 10	6, 6	4, F	5, 7	3, F
Copilot PS9	6, 8	6, 13	4, 6	3, F	5, 11	5, 8	2, 2	4, F	4, 15	3, F	4, 12	4, 5	6, 6	3, 4	5, F
DeepSeek PS2	2, 3	3, 4	2, F	3, 9	1, 5	2, 3	1, 10	3, 7	1, F	2, 6	1, 9	4, F	3, 4	4, 9	2, F
DeepSeek PS7	3, 7	4, 9	5, F	2, 8	3, 7	3, 5	2, 2	6, 12	4, 6	3, 11	4, 6	9, 14	1, 7	2, 3	3, F
DeepSeek PS9	3, 4	4, 13	4, F	3, 11	3, 4	1, 12	3, 6	3, 3	4, 7	3, 13	2, 9	6, 7	2, 10	4, 14	3, F

TABLE V

CODE METRICS FOR THE GENERATED USE CASE 15

	LoC	CpL	NDF	LpF	MCC
Claude PS2	553	0.08	4	87.75	16
Claude PS7	562	0.06	6	56.66	15
Claude PS9	499	0.06	4	75	18
Gpt PS2	400	0.07	6	42.16	10
Gpt PS7	292	0.05	6	32.66	7
Gpt PS9	572	0.13	9	37.55	19
Copilot PS2	472	0.09	6	50.6	21
Copilot PS7	524	0.11	6	59	30
Copilot PS9	555	0.09	6	61	29
DeepSeek PS2	527	0.04	5	62	17
DeepSeek PS7	544	0.05	5	63.8	17
DeepSeek PS9	592	0.02	6	62.5	19
Benchmark	478	0.04	4	75.5	9

Fig. 4. Heat map representing the average values of CrystalBLEU cross-similarity index for each couple of model-prompt strategy and the reference program

B. Generated Code Metrics

Each generated program is described by its values for the code metrics defined in the previous section. Table V presents, as an example, the metrics for UC15, the sole program that fails across all combinations after the fifteenth iteration. On average, the models produce code with greater LoC, Cpl, NdF, and MCC than the reference program, while LpF is lower.

The results indicate that GPT tends to generate code that is shorter, less complex, and more commented than the reference. For use case 10, which exhibits the lowest computational complexity, GPT is the only model that preserves a complexity value of 1 across all PSs, although its output fails tests at PS7. Claude likewise maintains complexity 1 at PS7 and produces code that passes the tests. Other models display higher complexity: Deepseek and Claude report values around 6 in some cases, while Copilot reaches up to 10. UC1, the only use case successfully completed by all models, yields generated-program complexity equal to or closely matching the benchmark contract. Measured complexity values generally range from 5 to 7, with the exception of GPT at PS9 that resulted in a complexity of 12.

C. Code Similarity

The CrystalBLEU similarity index, calculated for all pairs of source code generated for each use case (including the reference contract), enables evaluation of two aspects: intra-model similarity and inter-model similarity. Fig. 4 shows the heatmap of the average similarity indexes for all 15 UCs. The intra-model similarity index is an inverse indicator of a model's sensitivity to prompt. The results show that the similarity indexes computed among the three programs per use case generated by both DeepSeek and Claude are higher than those for the other two models, with average values ranging from 0.43 to 0.47. This indicates that the Claude and DeepSeek models are less sensitive to prompt quality. Conversely, GPT is the most sensitive to prompt strategy, with average intra-model similarity values between 0.26 and 0.27. Regarding inter-model similarity, the index calculated between DeepSeek and Claude are the highest, ranging from 0.32 to 0.36. We also observe low overall similarity between the reference code (the original program shown in the graph) and the code produced by the models. In this case, Copilot produces code least similar to the reference program, with values between 0.17 and 0.18.

D. Hallucinations in generated code

Table VI reports the count of hallucinations detected, by category, within the code generated at the end of the process. Analyzing hallucinations helps developers understand where models fail and intervene appropriately. Overall, we detected 24 Intent Conflicting, 32 Inconsistency, 2 Repetition, 34 Dead Code, and 56 Knowledge Conflicting instances. Although no model is immune to producing hallucinations, the analysis

TABLE VI
TOTAL NUMBER OF HALLUCINATIONS FOUND IN THE CODE PER TYPE. THE COLUMN *None* REPORTS THE NUMBER OF PROGRAMS GENERATED WITH NO HALLUCINATIONS.

	H1	H2	H3	H4	H5	None
Claude PS2	1	1	1	3	3	8
Claude PS7	1	2	0	4	3	8
Claude PS9	1	0	0	3	1	10
Gpt PS2	5	2	0	4	10	2
Gpt PS7	3	3	0	1	7	5
Gpt PS9	3	4	0	3	3	6
Copilot PS2	2	6	0	3	4	3
Copilot PS7	2	3	0	2	9	5
Copilot PS9	1	4	0	1	7	6
DeepSeek PS2	3	3	1	3	2	5
DeepSeek PS7	1	2	0	4	3	8
DeepSeek PS9	1	2	0	3	4	7

found that, overall, the Claude–PS combinations yield a larger number of programs free of hallucinations. Prompt strategies significantly affect results, especially for GPT and Copilot, and for all models they particularly influence the mitigation of Intent Conflicting (H1). PS7 and PS9 do not consistently affect the mitigation of Knowledge Conflicting (H5) issues arising from the use of deprecated libraries and modules. Comparing the presence of hallucinations with success and failure rates on test passing, we observe that Intent Conflicting (H1) completely prevents tests from passing (no program containing H1 passes the tests). The presence of H2 (Inconsistency) impacts test passing in 94% of cases.

V. CONCLUSION

This study enabled a systematic comparison of four freely accessible language models for generating Solana-Anchor smart contracts, for which we provided 15 use-case description from the repository Rosetta Smart Contracts, function signatures, and a test suite. The study highlighted substantial differences among the models across four factors: the ability to produce test-compliant code in the fewest iterations, the ability to generate code with satisfactory code metrics, sensitivity to prompt strategy, and the ability to mitigate hallucinations. Because hallucination identification relies partly on manual inspection, some degree of subjectivity is unavoidable despite the use of a structured taxonomy. Overall, the results show that Claude Sonnet 4.5 yields the best performance despite some rigidity with respect to prompt strategies. GPT-5 tends to produce less complex and, according to our structural metrics, better-structured code (while acknowledging that these metrics do not capture qualities such as maintainability, security, or idiomatic style), reacts more strongly to prompt-strategy variations, but is also the most prone to hallucinations and yields the fewest successes within 15 iterations. Copilot exhibits intermediate overall performance in both iterations and prompt-sensitivity while tending to produce more complex code. DeepSeek ranks just behind Claude in number of successfully completed use cases, absence of hallucinations, and low sensitivity to prompt strategies. Despite differing sensitivities, the three prompt strategies lead to different outcomes in iterations, success count, code quality, and hallucination rate. PS9, which includes all recommended techniques, performs best overall for success count and, for Claude and GPT, for required number of iterations. PS7 and PS9 reduce Intent-Conflicting hallucinations compared with PS2. However, these strategies do not fully mitigate Intent-Conflicting and Knowledge-Conflicting hallucinations: Intent conflicts persist because prompts can still produce logical errors in code, and Knowledge conflicts derive from deprecated libraries or APIs that cause test failures.

Applied to a less-common smart-contract language than Solidity, this study highlights several considerations for Anchor–Solana developers using generative models and supports future studies on AI-enhanced blockchain monitoring systems [13]. It also exposes performance differences among models and their sensitivity to prompt strategies, and it lays groundwork for further investigations, such as API-based usage of the models to evaluate sensitivity to configuration parameters and the usage of lightweight local models enhanced with RAG to assess performance of more sustainable setups.

REFERENCES

[1] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[2] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[3] R. Khojah, F. G. de Oliveira Neto, M. Mohamad, and P. Leitner, "The impact of prompt programming on function-level code generation," *IEEE Transactions on Software Engineering*, 2025.

[4] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, L. Zhang, Z. Li, and Y. Ma, "Exploring and evaluating hallucinations in llm-powered code generation," *arXiv preprint arXiv:2404.00971*, 2024.

[5] C. Laneve, A. Spanò, D. Ressi, S. Rossi, and M. Bugliesi, "Assessing code understanding in llms," in *Formal Techniques for Distributed Objects, Components, and Systems*, C. Ferreira and C. A. Mezzina, Eds. Cham: Springer Nature Switzerland, 2025, pp. 202–210.

[6] N. Petrović and I. Al-Azzoni, "Model-driven smart contract generation leveraging chatgpt," in *International conference on systems engineering*. Springer, 2023, pp. 387–396.

[7] E. A. Napoli, F. Barbàra, V. Gatteschi, and C. Schifanella, "Leveraging large language models for automatic smart contract generation," in *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2024, pp. 701–710.

[8] G. Baralla, G. Ibba, and R. Tonelli, "Assessing github copilot in solidity development: Capabilities, testing, and bug fixing," *IEEE Access*, 2024.

[9] ——, "Reasoned or rapid code? unveiling the strengths and limits of deepseek for solidity development," *Information and Software Technology*, p. 107917, 2025.

[10] M. Bartoletti, L. Benetollo, M. Bugliesi, S. Crafa, G. Dal Sasso, R. Pettinau, A. Pinna, M. Piras, S. Rossi, S. Salis *et al.*, "Smart contract languages: A comparative analysis," *Future Generation Computer Systems*, vol. 164, p. 107563, 2025.

[11] A. Eghbali and M. Pradel, "Crystalbleu: precisely and efficiently measuring the similarity of code," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[12] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" *arXiv preprint arXiv:2310.06770*, 2023.

[13] D. Ressi, R. Romanello, C. Piazza, and S. Rossi, "Ai-enhanced blockchain technology: A review of advancements and opportunities," *Journal of Network and Computer Applications*, vol. 225, p. 103858, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804524000353