On the Use of LLMs for Upgrading Legacy Smart Contracts: An Initial Study

Andrea Pinna[1,*], Gavina Baralla[1], Giacomo Ibba[1] and Roberto Tonelli[1]

[1]Department of Mathematics and Computer Science, University of Cagliari, Via Ospedale 72, Cagliari

Abstract

Legacy smart contracts, which were developed using outdated versions of programming languages, present significant challenges in terms of maintainability, security, and compatibility. These challenges arise from substantial changes in syntax and best practices that have been adopted in more recent versions to enhance safety and security. Notably, programming languages for smart contracts evolve at a faster pace than traditional programming languages, making timely upgrades even more critical. Upgrading these contracts to the latest version of the language is essential for mitigating known vulnerabilities, leveraging improved security features, and ensuring compatibility with contemporary blockchain environments. Large Language Models have demonstrated considerable utility in the realm of automatic code generation, thereby accelerating the development process for programmers. This paper investigates the application of LLMs for the purpose of upgrading smart contract code. In this preliminary study, we specifically examine the effectiveness of the LLM Claude 3.7 in automating the migration of legacy Solidity smart contracts to version 0.8.20 of the language. Through a series of controlled experiments, we assess Claude 3.7's performance in generating syntactically correct, secure, and functional upgraded versions of a benchmark set comprising 21 selected legacy Solidity source codes, which are representative of common use cases for smart contracts. The experimental design, which includes prompt engineering and dataset selection, aims to obtain both quantitative measurements and qualitative assessments of the modifications made to the code, the generated test suite, and the auto-generated technical reports, as well as the overall effectiveness of the approach. The results of the analysis indicate significant variability in the performance of the LLM across the tasks, particularly in relation to the varying levels of complexity inherent in the legacy code. This trend is further substantiated by multiple analyses, including the number of iterations required to achieve a compilable result free of errors during testing, the ability to manage outdated or deprecated practices in Solidity programming, and the depth of detail provided in the generated technical reports. This study is intended to serve as a precursor to a broader investigation that will compare different LLMs in the upgrading of contracts written in various programming languages for smart contracts.

Keywords

Blockchain, Smart contracts, Upgrade, Large Language Models, Solidity, Claude Anthropic

1. Introduction

Blockchain smart contracts, which are programs written to be executed within a blockchain, were originally conceived to implement immutable and verifiable agreements between parties[1]. This immutability makes it particularly important to determine the presence of vulnerabilities in the code before deploying the contract, using appropriate tools, and in any case, to follow best practices and use the most up-to-date versions of the language and compiler[2, 3, 4]. Failing to undertake this activity can result in security risks and asset losses[5]. It is well known that smart contracts constitute the blockchain component of all decentralized applications (dApps), and it is crucial that when updating the blockchain component, compatibility with the rest of the system is not lost. At the same time, delaying updates can lead to higher costs in the long term[6]. In the field of programmable blockchains, it is not uncommon for a smart contract code to become "legacy" in a short period of time. Indeed, in this sector, there are frequent updates to the languages and supporting libraries, rendering the code outdated

DLT2025: 7th Distributed Ledger Technology Workshop, June, 12-14 2025 - Pizzo, Italy

*Corresponding author.

pinna.andrea@unica.it (A. Pinna); gavina.baralla@unica.it (G. Baralla); giacomof.ibba@unica.it (G. Ibba); roberto.tonelli@unica.it (R. Tonelli)

0000-0002-7530-0521 (A. Pinna); 0000-0002-3119-9388 (G. Baralla); 0000-0003-3087-1969 (G. Ibba); 0000-0002-9090-7698 (R. Tonelli)

and incompatible, even within a few months. Therefore, while could be possible to update blockchain smart contracts (either natively or by using specific patterns like the proxy pattern)[7] maintaining compatibility and preventing disruption to the overall system is a crucial concern when updating the core blockchain components of a dApp.

This underscores the need for scalable, reliable, and secure methods to upgrade these contracts. Several methodologies have been proposed to address the problem of smart contract upgrades, ranging from manual code audits and patching, to automated bytecode rewriting and pattern-based upgrade models [8, 9]. Despite their effectiveness, these approaches share common limitations, such as requiring significant manual efforts, specialized technical knowledge, and the risk of introducing further vulnerabilities or breaking compatibility during the upgrade process. In this context, automated methodologies based on advanced artificial intelligence techniques, such as Large Language Models (LLMs), have emerged as promising alternatives capable of bridging these gaps[10]. The proven effectiveness of LLMs in Solidity programming tasks has set a significant precedent for recognizing their potential in smart contract development [11, 12, 3]. This capability is in line with the vision of making smart contract upgrades more accessible and scalable [9]. The types of issues in legacy contracts that LLMs could help address during the upgrade process include specific vulnerabilities such as function selection conflicts and storage slot collisions [13, 4].

In this work, we explores and evaluate the application LLMs in automating the upgrade of legacy smart contract code. For this initial study, we focus on the upgrade of legacy Solidity code to a modern language version, namely Solidity 0.8.20 and focus our attention of Claude 3.7, an advanced LLM developed by Anthropic [1] The selection of Claude 3.7 is underpinned by its demonstrated ability to prioritize security issues in code generation, an essential feature when dealing with the inherent vulnerabilities in legacy contracts [3].

Our research is structured into three distinct phases. The first phase involves the creation of a dataset comprising 21 legacy smart contracts, carefully selected to represent common use cases within the Solidity ecosystem. This dataset serves as the foundation for our experimental investigations. In the second phase, we engage in prompt engineering to define specific prompts that guide the model's tasks, establishing a systematic workflow for conducting experiments with Claude 3.7. This ensures clarity in the objectives and tasks assigned to the model, facilitating a structured approach to the evaluation process. The final phase consists of executing the experiments and analyzing the results. We focus on assessing the model's performance in generating upgraded code, examining the variation in lines of code (LoC), and evaluating the quality of the auto-generated reports. This structured approach not only allows for a thorough evaluation of Claude 3.7's capabilities but also lays the groundwork for future research aimed at comparing multiple LLMs and expanding the scope to include a broader range of programming languages and smart contract scenarios.

The structure of this paper is as follows: Section 2 examines previous applications of smart contract upgrades and the use of LLMs in smart contract development, identifying gaps that this study aims to address. Section 3 describes the methodology, including the experimental design used to evaluate the performance of Claude 3.7 in upgrading legacy smart contracts. Results and analysis are presented in Section 4, which provides a thorough assessment of both the process and the model's performance. The Discussion section interprets these findingsand the Conclusions section summarizes the study's contributions.

2. Related Work

In recent years, significant research efforts have explored the application of large language models within the domain of smart contract development, particularly focusing on automating contract generation, validation, vulnerability detection and upgrade processes.

Zhao et al. [3] proposed SCCLLM, a novel and promising approach that leverages both LLMs and in-context learning for automatic smart contract comment generation. Their method, consisting in

[1] https://www.anthropic.com/

two-phase strategy, considers semantic, syntactic, and lexical information to retrieve relevant examples from a historical corpus. These examples are then used as demonstrations for in-context learning with ChatGPT, enabling the model to generate high-quality comments without parameter updating.

Barbàra et al. [10] investigated the feasibility of using LLMs to generate production-ready Solidity smart contracts for non-technical users. Using a lease agreement as a case study, they employed GPT-4 with different prompt designs following the CO-STAR methodology. While 94.1% of generated contracts compiled successfully and most showed only low-impact vulnerabilities in automated tests, expert analysis revealed critical logical flaws that automated tools couldn't detect. The researchers found that current LLMs are incapable of generating production-ready smart contracts, highlighting the significant gap between syntactically correct code and functionally sound implementations.

Chatterjee and Ramamurthy [5] conducted a comprehensive evaluation of various Large Language Models (LLMs) for generating Solidity smart contracts on the Ethereum blockchain. Their methodology involved testing the generated contracts for accuracy, efficiency, and code quality using both descriptive and structured prompting techniques across three contract scenarios of increasing complexity. Their findings revealed that all models struggled with more complex implementations and most LLMs overlooked critical security considerations unless explicitly prompted. The authors concluded that current LLMs show promise for adapting existing smart contracts or assisting developers but are not yet suitable for industrial smart contract generation due to security and efficiency concerns.

Napoli et al. [13] evaluated four leading LLMs (GPT-4-Turbo, Claude-3.5-Sonnet, Mistral-Large, and Gemini-1.5-Pro) for automated smart contract generation from legal agreements. Their framework assessed functional completeness and security across five agreement types using eleven design patterns. Results showed Claude and GPT-4-Turbo significantly outperforming other models, though all generated contracts contained security vulnerabilities. While LLMs demonstrated promising capabilities in code generation, they concluded that current models require substantial human oversight for production-ready smart contracts, particularly for complex agreements.

Boi et al. [4] propose using fine-tuned Large Language Models for smart contract vulnerability detection. They fine-tuned Llama-2-7b-chat-hf on a dataset of smart contract vulnerabilities, creating a unified mapping between OWASP and SWC classifications. Their model achieved 59.5% accuracy across vulnerability types, performing especially well on arithmetic vulnerabilities (93.3%). Though not outperforming specialized tools like Mythril, their approach offers greater accessibility to non-security experts and provides contextual remediation advice, representing a promising direction for integrating LLMs into blockchain security workflows.

Baralla et al. [14] assessed GitHub Copilot for Solidity development, examining its capabilities in code generation, implementation assistance, vulnerability detection, and unit testing. Results showed Copilot excels with simple contracts but struggles with complex logic and security patterns. While beneficial for standard implementations, Copilot requires human oversight for security-critical smart contract development.

Karanjai et al. [12] presented SolMover, a dual-LLM framework for translating Solidity smart contracts to Move language. Their approach combines concept mining through retrieval-augmented generation with subtask-based code production, enhanced by iterative compiler feedback. SolMover significantly outperformed single-model approaches, successfully translating 54.6% of contracts versus 31.2% for GPT-3.5, demonstrating that LLMs can effectively generate code in low-resource languages with minimal fine-tuning.

Ibba et al. [15] developed a methodology using Claude 3.5 and GPT-4 to generate synthetic Ethereum smart contracts with Denial of Service vulnerabilities. Their research showed Claude outperformed GPT-4, requiring fewer prompts and producing higher quality outputs. The study addressed the lack of training data for machine learning security tools by creating realistic vulnerable contracts through structured prompt engineering. This approach enables better development of classification and anomaly detection models for blockchain security.

However, despite these advances, research on the use of LLMs to upgrade legacy smart contracts remains limited. Most studies focus on generating new contracts or analyzing vulnerabilities, leaving a gap in the literature regarding the application of LLMs to facilitate the upgrade and migration of

existing contracts. This highlights the need for further research to explore how LLMs can be effectively used to modernize and improve legacy smart contracts while ensuring data security and integrity.

3. Methodology

This study employs a methodology based on controlled experiments to evaluate the effectiveness of using LLMs for the automatic upgrading and testing of smart contracts written in legacy programming languages. Additionally, it aims to assess the overall efficacy of this approach. As a preliminary investigation, this work focuses on a single recently released LLM (Claude 3.7 Sonnet), and the evaluation of both the results and the methodology will provide a foundation for future, more extensive studies.

The research methodology is organized into three distinct phases. The first phase involves creating a dataset of legacy smart contracts. The second phase focuses on defining prompts and the experiments workflow. The third phase consists of executing the experiments. The last phase entails analyzing the results and evaluating the effectiveness of the approach.

In the following sections, we first describe the creation and composition of our experimental dataset, which consists of legacy smart contract source codes. Next, we outline the characteristics of the selected LLM system, including its limitations and our strategies for effective utilization. Subsequently, we provide a detailed account of the iterative process employed in designing, optimizing, and refining the prompts to accurately guide the model in generating both upgraded source codes and the corresponding test suites. Finally, we present the workflow of the experiments, detailing each step of our systematic evaluation to minimize potential biases.

3.1. Experimental dataset

To evaluate the effectiveness of LLMs in upgrading legacy source code, it is essential to have a heterogeneous and representative set of contracts that encompasses both the types of Use Cases and the legacy versions of the programming language. However, given the preliminary nature of this work, the objective is to maintain a limited dataset. designed to provide an initial representative sample while minimizing complexity and ensuring the preliminary study remains manageable in scope.

In constructing the dataset, we focus exclusively on smart contracts written in Solidity and considered two sources: Etherscan verified contracts and the official Solidity documentation. The selection criteria for the source codes are based on the need to include various types of use cases and, for each use case, different legacy versions of the programming language. As a result of the selection, 21 source codes were selected to compose the experimental dataset. Six of these—Ballot, SimpleAuction, BlindAuction, Purchase, ReceiverPays, and SimplePaymentChannel—were sourced from the official Solidity documentation [2]. The remaining contracts were collected from Etherscan and their corresponding Ethereum addresses are listed in Table 1.

As reported in Table 2, the set of Use Cases represented in this dataset includes ERC20 tokens, custom token implementations (with additional logic) , auctions, voting mechanisms, crypto transfers, time locks, and vesting schedules. Many contracts in the dataset incorporate the SafeMath library to handle arithmetic operations safely, preventing problems such as overflows and underflows, which were critical vulnerabilities in previous versions of Solidity before automatic checks were implemented in version 0.8.0. Pragma versions, namely the version of the solidity compiler specified inside the source file, range from 0.4.16 to 0.6.12, reflecting a cross-section of syntax rules, functions, and keywords that have been deprecated in the current version of the language. Source codes vary in size and structural complexity, with lines of code ranging from fewer than 50 to over 500, and function counts from 3 to more than 50.

In order to evaluate a potential relationship between program complexity and the quality of the work performed by the LLM, we categorize the source codes into three levels of complexity: low, medium, and high.

[2] https://docs.soliditylang.org/en/v0.5.5/solidity-by-example.html

Table 1
Smart contracts source detail and classification

File Name	Address/source	Classification
Alcanium	0a3ab1d51acc4d8fd8494b211949f707794ca3b6	Medium
BNIToken	00a4A3e9279678CA1b564Bca66396bb5801192dA	Medium
FreePalestine	000a3f282f082aa9cf95271a97a0515b652daa6b	Medium
NanaInu	0a1bdb38e14e9c4c1eaddce81be08779d5c200d2	Medium
Omosubi	ed2f4dae9efa6da1f6b78db7c3959c1226956847	Medium
O2OToken	ed00a2cba066714999ec703350e0a5b6b7ab66cb	Medium
Shop	Ff0ab053B98E57E9cDFa2E6Be2C3aec69AE91533	Hight
TokenMintERC20Token	00a1f48b91b794ff3da36f97be48ebda5abcbeb3	Hight
Eloncat	88af27d10123a35b15e3fa1400e6470bc6e28925	Hight
Entropy	0a0e3bfd5a8ce610e735d4469bc1b3b130402267	Hight
PonderAirDropToken	88e7029a16443ef0491c5f46042d34d0c56e691f	Hight
BlindAuction	docs.soliditylang.org	Low
SimpleAuction	docs.soliditylang.org	Low
Ballot	docs.soliditylang.org	Low
EtherTool	ed7c4a8ff24c46333e1afbe6740ff6e77c6f3de4	Medium
Purchase	docs.soliditylang.org	Low
ReceiverPays	docs.soliditylang.org	Low
SimplePaymentChannel	docs.soliditylang.org	Low
SepukuToken	0x4a8f506bbf946f030d3bb4c861db14a252d3c5cf	Medium
Timelock	ff2df283f8b91919d3c1b08031c1846b80f0deb3	Low
IcoLib	ed045abcd09fa8223d06bca4b0ad562a1c467f10	Hight

Table 2
List of smart contracts selection

File Name	Pragma	Category	Lines of Code	Contract declarations	Total n. of functions
Alcanium	^0.5.0	ERC20	72	3	16
BNIToken	0.4.18	ERC20	111	7	19
FreePalestine	0.6.6	ERC20	108	4	21
NanaInu	^0.5.0	ERC20	72	3	16
Omosubi	0.6.0	ERC20	198	3	19
O2OToken	^0.4.16	ERC20	208	6	18
Shop	^0.4.18	ERC20 and Transfer	314	8	29
TokenMintERC20Token	^0.5.2 (earliest)	ERC20	429	4	29
Eloncat	^0.6.12	Custom ERC20	545	6	54
Entropy	^0.5.16	Custom ERC20	517	2	31
PonderAirDropToken	^0.4.21	Custom ERC20 and Transfer	498	4	33
BlindAuction	>0.4.23 <0.6.0	Auction	96	1	5
SimpleAuction	>0.4.22 <0.6.0	Auction	52	1	3
Ballot	>0.4.23 <0.6.0	Voting	79	1	5
EtherTool	^0.4.18	Transfer	144	2	19
Purchase	>0.4.22 <0.6.0	Transfer	70	1	3
ReceiverPays	>=0.4.24 <0.6.0	Transfer	41	1	5
SimplePaymentChannel	>=0.4.24 <0.6.0	Transfer	61	1	7
SepukuToken	^0.5.0	ERC20	149	2	27
Timelock	^0.4.24	TLC	49	3	6
IcoLib	0.4.24	Vesting	477	6	47

Low-complexity files with fewer than 100 lines of code and fewer than 10 functions, often focusing on a single functionality with minimal internal contracts. *Medium-complexity* files with 100 to 300 lines of code and containing 10 to 30 functions, may also involve moderate multi-contract structures or custom logic. *High-complexity* files exceed 300 lines of code or 30 functions, often characterized by large codebases, advanced functionality, and multiple internal contracts. As reported in Table 1, of the 21 selected contracts, 7 were classified as low complexity, 8 as medium, and 6 as high complexity.

Despite the relatively small number of contracts included, the diversity of these contracts provides a preliminary foundation for assessing the LLM's ability to manage upgrades, maintain interface consistency, and address deprecated features.

3.2. LLM Interaction

We have chosen to use a single LLM to consolidate our approach and facilitate future studies comparing the performance of different LLMs. The LLM selected for this study is Claude 3.7 Sonnet, released in February 2025 [3].

Comparative studies have positioned Claude, alongside GPT-4-Turbo, as superior in handling complex coding tasks, supporting our choice for this initial research. The performance hierarchy noted in these studies, where Claude excels at both syntactic correctness and managing programming complexity, firmly aligns with the need for effective and secure smart contract upgrades [13].

This model operates within specific computational constraints that affect user interaction patterns. The model uses a resource allocation system where conversation length directly impacts available interactions—approximately 45 messages every 5 hours for shorter conversations (about 200 English sentences of 15-20 words each), decreasing to approximately 15 messages when processing larger documents. This limitation arises from the model's architecture, which processes the entire conversation history, including attachments, with each new query.

Users experience two notable interaction artifacts: first, the need to prompt the model with "continue" commands when responses exceed output limits, resulting in fragmented information distribution; and second, system notifications warning that "Long conversations consume usage limits more quickly."

When the output limit is reached, we observed that in code generation, using the "continue" command often results in trivial errors, including syntactic mistakes. This occurs because the LLM modifies the code it generated in the previous iteration. We frequently found instances where lines of code were not deleted correctly or where code was missing. Empirically, we discovered that if the "continue" command is issued promptly (within a few seconds after the message appears), the quality of the edited code improves significantly, particularly in terms of syntactic errors and missing code, especially for longer code segments (over 300 lines)
[4].

These constraints reflect the resource-intensive nature of maintaining and processing large contextual information. Usage limits for Claude Pro (approximately five times higher than the free service, which is the version used for our experiments) are reset every five hours. For optimal use, specific precautions are necessary, such as starting new conversations for separate topics, grouping multiple questions into single messages, and avoiding redundant file uploads to maximize available computing resources.

3.3. LLM Prompt Structuring

A key element of this study is the development of an effective prompt to assess the performance of the LLM. The purpose of the prompt is to provide the LLM with the necessary description of the objectives and information to carry out the desired process. For the objectives of this study, the LLM is tasked with producing upgraded Solidity code from the legacy code and generating tests to verify its correct functionality. Additionally, at the end of this process, a report is requested to explain the actions taken.

The methodology used for defining the prompt involves refinement through successive iterations. Given the complexity of the process of upgrading legacy Solidity smart contracts, considerable effort was spent on iteratively refining the initial prompt before arriving to the final prompt. At each iteration, contextual information about the local execution environment was added to the prompt. This included details such as IDE specifications, the version of the Hardhat testing framework, Solidity compiler versions, and local testnet configurations. This contextual embedding aimed to maximize the clarity and relevance of the outputs generated by the LLM.

The adopted prompt engineering approach consists of dividing the process into two distinct steps:

- **Step 1: Contract Upgrade and Testing**
  Upgrade: To upgrade a deployed legacy Solidity contract to a newer Solidity version (specifically,

[3] https://www.anthropic.com/news/claude-3-7-sonnet

[4] We emphasize that this is an empirical result that should be further investigated; however, we observed a potential phenomenon of temporary caching or state retention that appears to degrade over time between continuation prompts

version 0.8.20), Claude was prompted with the key requirement to ensure that external interfaces and function declarations remained the same between legacy and upgraded contracts.
Test Suite Generation: In the same prompt, the LLM was instructed to generate a JavaScript test suite that would work with both the legacy and upgraded versions of the contract. The test suite must utilize the Hardhat framework and be structured to test semantic consistency across versions without modification. Specific design requests were incorporated into the prompt, including:

  - Using an environment variable to select the smart contract to test.
  - Using fully qualified contract naming for precise targeting.
  - Using an automatic Solidity version detection mechanism.
  - Include a conditional logic in test cases to accommodate internal behavioral differences between various Solidity versions

- **Step 2: Report Generation**
  Upon successful compilation and passing of all unit tests, a second prompt requested the LLM to produce a detailed technical report summarizing the upgrade results. The second prompt specify that the report must included:

  - A concise changelog identifying syntax modifications and new language features introduced by upgrading to Solidity 0.8.20.
  - A quantitative summary comparing the number of implemented functionalities (before and after upgrading).
  - A verification that functionalities of the legacy and upgraded contracts retained identical interfaces and external behavior.
  - The comprehensive assessment of test coverage, detailing the number of functionalities tested in each version of the contract.
  - An analysis of the necessary test adaptations made to address the differing behaviors in the Solidity versions of the legacy and upgraded contracts, including distinctions in error handling.
  - A summary of the most significant modifications that impact contract security, efficiency, and maintainability.

The decision to separate the prompt into two steps is based on the following considerations. *Iterative nature of the process*: Upgrading smart contracts and developing corresponding test suites is inherently an iterative process. Critical issues or requirements for changes may become apparent only after the test protocols have been executed, necessitating post-implementation changes to the upgraded contract. *Greater accuracy of documentation*: A report generated following the successful completion of the contract upgrade and testing phases will inherently demonstrate greater accuracy, effectively reflecting all changes implemented during the iterative development cycle. *Prioritization of primary objectives*: The initial prompt should maintain focus on the primary activities, namely the contract upgrade and test suite development. Including excessive requirements risks reducing the effectiveness with which the primary objectives are addressed. *Empirical basis for analysis*: A subsequently generated report may be based on empirical test results rather than theoretical projections, thus providing a more substantial basis for evaluating the effectiveness of the upgrading process. *Limitation of Claude chat length*: Claude models impose constraints on the length of individual outputs, often requiring the use of continuation prompts for extensive responses.

The resulting prompts are reported as follows.

Prompt for upgrading and testing
Upgrade this smart contract written in Solidity to the latest pragma version (0.8.20), ensuring that the function interfaces remain identical. The project is structured with two separate contract folders ('legacy' and 'upgraded') where contracts maintain the same name in both folders but use different compiler versions. Develop a single set of unit tests in a .js file that can be used to verify the semantic correctness of both contract versions without any modifications. The tests must be compatible with both versions since the function declarations are the same, but should account for any changes in internal behavior or error mechanics between Solidity versions. The project is configured with the following package.json dependencies:

```
"devDependencies": {
"@nomicfoundation/hardhat-toolbox": "^5.0.0",
"hardhat": "^2.22.19"
}
```

Structure the tests using the following pattern:

- Implement an environment variable configuration (CONTRACT_VERSION) to control which contract version to test
- Use fully qualified contract names to reference the specific contract files
- Create a version detection mechanism that automatically adapts tests to the appropriate Solidity version
- Use the current Ethers v6 deployment pattern with waitForDeployment() instead of deployed()
- Include conditional test logic that can handle different error mechanisms between Solidity versions.

The test file should follow this general structure:

```
describe("Contract Tests", function () {

   // The full specification is publicly available on GitHub
   // at https://github.com/LLM-and-blockchain/Solidity-Claude/blob/main/prompts.md

});
```

Each test should verify that both versions of the contract maintain identical behavior from a user perspective, while adapting to internal implementation differences between Solidity versions.

Prompt for report generation
Based on the implementation of both the legacy and upgraded smart contracts and their corresponding test results, generate a comprehensive technical report that includes:

1. A detailed changelog documenting all modifications between the legacy and upgraded contract versions, including:
    - Syntax and language feature updates necessitated by the Solidity version change
    - Security improvements and vulnerability mitigation
    - Gas optimization techniques applied
    - The technical rationale justifying each significant change
2. A quantitative analysis of functionality, including:
    - Total count of functions implemented in the legacy contract
    - Total count of functions implemented in the upgraded contract
    - Any new capabilities introduced in the upgraded version
    - Verification that all legacy functionalities remain accessible through identical interfaces
3. A test coverage assessment:
    - Count of functionalities successfully tested in the legacy version
    - Count of functionalities successfully tested in the upgraded version
    - Description of any version-specific test adjustments required: if referred to the contract code implementation or if referred to test code syntax
    - Analysis of edge cases and how they were handled differently between versions
4. A concise executive summary highlighting the most significant changes and their impact on contract security, efficiency, and maintainability.

The full prompts can be accessed in the GitHub repository of this research[5].

3.4. Experimental workflow

The experimental workflow begins by examining each legacy smart contract file one by one and ends with the production of a dataset of upgraded smart contracts and the reports generated for each upgrade. A representation of the experimental workflow is illustrated in Fig. 1.

To ensure the process of upgrading each contract remained independent and to avoid bias, each experiment involves a new and separate chat of Claude 3.7 LLM.

Each conversation with the LLM begins with the first step of the prompting process described above, which includes uploading the legacy contract source and a written request to upgrade the legacy code to Solidity 0.8.20, ensuring that interfaces are preserved and generating the corresponding test suites.

Once Claude 3.7 produces the upgraded contract and the related test suite, these outputs are compiled and executed locally in the environment described below.

Whenever errors occurred, either during compilation or test execution, the error report displayed in the terminal is directly copy and pasted into the running chat to minimize any additional context that

[5]https://github.com/LLM-and-blockchain/Solidity-Claude

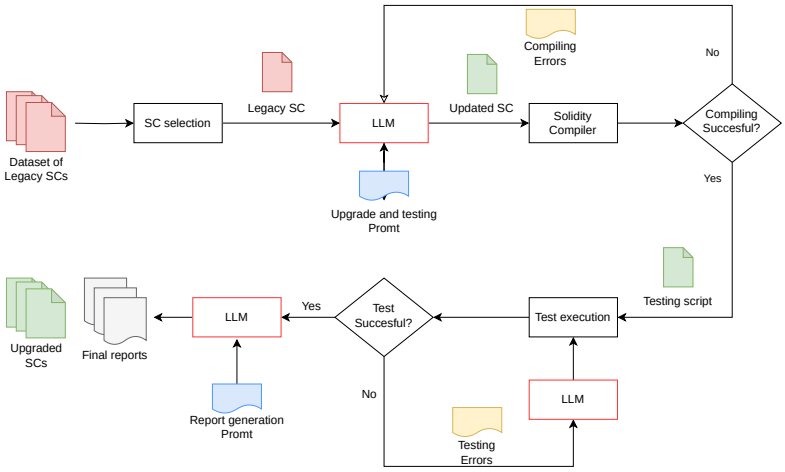

Figure 1: Representation of the workflow for the experiment conducted on each file in the dataset of legacy smart contracts. The two-step prompting process is illustrated in light blue, while feedback from the local environment is shown in light yellow.

could bias the LLM. This iterative process led to resolving syntactic and semantic errors, ensuring the correct code for the upgraded contract and robust test suites.

When the upgraded contract compiles successfully and the test suite passes all checks, Claude 3.7 is provided with the second-step prompt detailed above. This prompt requests the generation of a detailed technical report about the produced code and the iterations required to achieve a fully functional upgrade. This final step ensures that each upgrade is thoroughly documented, providing valuable insights for interpreting the results.

3.4.1. Experimental setup

The primary frameworks and tools utilized for experiments include the LLM Claude 3.7, accessed remotely via the web interface, as well as a local environment consisting of the following components: Visual Studio Code IDE (version 1.98.2), an EVM testnet based on Ganache (version 7.9.2), and the Hardhat framework (version 2.22.19) for compiling and testing smart contracts. Additionally, the environment includes Node.js (version 20.12.2) and Web3.js (version 1.10.0) to ensure compatibility and facilitate efficient interaction with Ganache. Multiple versions of the Solidity compiler were maintained, providing the flexibility to switch between versions as needed for compiling both legacy and upgraded smart contracts. The local environment runs on an Asus ExpertBook powered by a 12th Gen Intel(R) Core(TM) i7-1265U processor running at 1.80 GHz, complemented by 40 GB of installed RAM (39.7 GB usable). The system operates on Windows 11 Pro in a 64-bit x64 architecture.

3.5. Evaluation of results

The results of the experiments are evaluated in both quantitative and qualitative terms.

In terms of quantitative aspects, the focus is on the number of iterations required to obtain a compilable code and to pass the tests. This data provides a clear indication of the efficiency of the upgrade process.

For the qualitative assessment, a manual inspection is conducted to compare the produced upgraded Solidity code with the legacy code, also in relation to what described in the generated report. This inspection enables the evaluation of the entire set of experiments and reveals whether the LLM consistently adopts the same strategy to solve a specific problem or if its approach varies depending on the changes in the examined source code

Table 3
Number of further iterations with the LLM for each experiment before obtaining a compiling code and a successfully test suit.

Contract file	# It. for compiling	# It. for test	Additional prompt	# continue
Alcanium	0	0	0	0
BNIToken	2	0	0	0
FreePalestine	0	2	0	0
NanaInu	0	0	0	0
Omosubi	0	1	0	1
O2OToken	4	2	1 - add test missing code	4
Shop	2	7	3 - fix errors in test file - Add other test - add missing code	5
TokenMintERC20Token	1	1	0	0
Eloncat	2	2	2 - add test missing code - fix the warning	2
Entropy	0	2	1 - add test missing code	1
PonderAirDropToken	2	4	3 - fix the warning - add test missing code -I cannot modify legacy contract	5
BlindAuction	0	2	0	0
SimpleAuction	0	2	0	0
Ballot	0	1	0	0
EtherTool	0	2	0	4
Purchase	0	1	0	0
ReceiverPays	1	2	0	0
SepukuToken	1	3	0	0
SimplePaymentChannel	1	1	0	0
Timelock	0	2	0	0
IcoLib	1	3	2 - fix the warning - add test missing code	2

4. Results and Analysis

In this section, we analyze three types of results: the performance of the process, the metrics of the upgraded contracts, and finally, the analysis of the auto-generated reports. The experimental results and the dataset are available on the Github repository of this research[6].

4.1. Process analysis

The initial evaluation of Claude 3.7's performance in generating upgraded Solidity code and its corresponding tests is quantitative. It is based on the number of iterations required to produce compilable code using version 0.8.20 of the Solidity compiler and to ensure the successful execution of the tests.

The Table 3 summarizes these data and also includes instances where additional prompts were necessary (beyond simply copying and pasting errors encountered during compilation or test execution) to guide the language model toward the desired outcome. Additionally, it notes the number of times the prompt "continue" was required to address the output length limitation of Claude 3.7, as discussed earlier.

We can observe that in just over half of the experiments (11 out of 21), the number of iterations required for compilation is zero, meaning the LLM produced directly compilable code with the specified version of the compiler. In 5 cases, only one additional iteration was needed. In four cases—specifically BNItoken Shop, Eloncat, and PonderAirDropToken—two additional iterations were necessary. In one

[6]https://github.com/LLM-and-blockchain/Solidity-Claude

Table 4
Iteration statistics for different complexity Levels of legacy smart contract source code.

	Compiling It.	Testing It.	Total n. of It.	n. of Continue
Low Complexity				
Average	0.2857	1.5714	1.8571	0
St. Dev.	0.4879	0.5345	0.6901	0
Medium Complexity				
Average	0.4286	1.1429	1.5714	0.7143
St. Dev	0.7868	1.2150	1.3973	1.4960
High Complexity				
Average	1.7143	3.0000	4.7143	2.7143
St. Dev	0.9832	2.0000	2.4976	1.9760

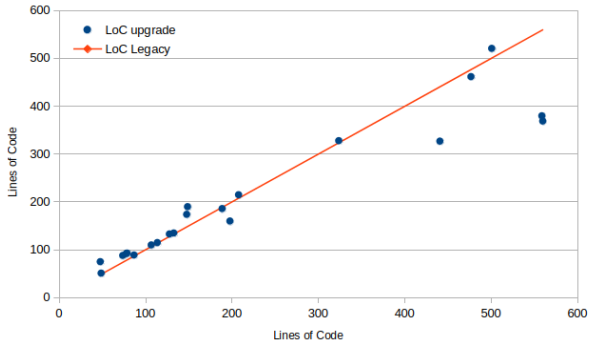

Figure 2: Dispersion diagram of the number of Lines of Code after the upgrade in comparison with the original number of LoC (orange line).

case, Omosubi, four iterations were required before the LLM produced compilable code.

Regarding the generation of the test suite, it is noteworthy that only in three cases did the tests work successfully on the first output. In most cases, one or two iterations were needed. In one instance, four iterations were required for PonderAirDropToken, and seven for Shop. These two experiments also required more additional prompts, as well as error messages, as detailed in the table.

Finally, regarding the number of times it was necessary to send "continue" to the LLM, it is noteworthy that this operation was not required in over half of the cases (13 out of 21).

Analyzing the statistics presented in the Table 4, which are categorized according to the complexity criteria defined in the previous section, we can observe that the statistics related to the number of iterations needed to achieve correct compilation tend to increase with the complexity of the legacy code. This trend is particularly pronounced in the high complexity category, where the average number of iterations is approximately six times that of low complexity and four times that of medium complexity.

Similarly, for the iterations required to obtain successfully passing tests without errors, the high complexity category also shows an average that is about twice that of both low and medium complexity. When considering the total number of iterations, it is evident that low complexity has a slightly higher average (about 17%) compared to medium complexity, while high complexity sources require, on average, more than two and a half times the number of iterations needed for the other two categories.

Finally, it is interesting to note that the average number of "continue" commands needed to obtain the desired output is zero for all contracts in the low complexity category, approximately 0.7 for contracts in the medium complexity category, and 2.7 for contracts in the high complexity category.

It is worth noting that the standard deviations are all high relative to the mean, primarily due to the small number of experiments and the heterogeneity of the contracts examined.

4.2. Analysis of generated upgraded contracts

The second analysis conducted involves a quantitative evaluation of the variation of LoC and a manual inspection of the upgraded code produced.

Overall, on average, the LLM extends upgraded contracts by 2.2 percent of LoC with respect to legacy

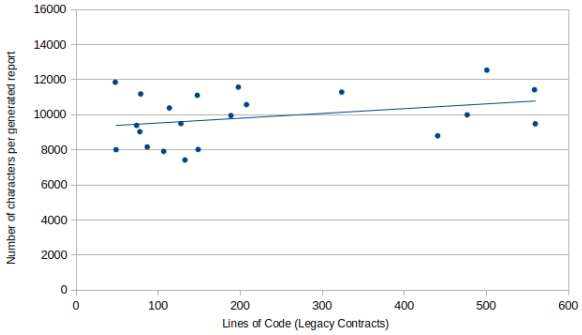

Figure 3: Number of characters of the auto-generated upgrade report versus the number of Line of Code of the legacy contract. The line represent the linear regression.

ones. However, when contracts are grouped by complexity, it is observed that low-complexity contracts are extended on average by 11.52%, while medium complex contracts are extended by 7.86%. In contrast, highly complex contracts are, on average, shortened by 12.62% in terms of the number of lines of code. That variations are represented in Fig. 2.

The automatic upgrades of legacy Solidity smart contracts, performed by the Claude 3.7 LLM, aimed to maintain the same interface as the legacy contracts while implementing various modifications to meet the syntax rules of Solidity 0.8.20, as well as to enhance safety and security. A summary of the manual inspection is reported in Table 4.2.

Overall, the upgrade process corrects the syntax of keywords used to define constructs (such as `abstract` and `interface`) and functions (like `override`), as well as built-in modifiers (including `external`, `internal`, and `view`). The upgrade consistently replaces the deprecated `.now` with `block.timestamp` and always adds messages to `require` statements when they are missing. In two instances, it retains `assert` statements without accompanying comments. Additionally, it almost always removes the visibility of the constructor, leaving it intact in only one case.

A significant concern is the inconsistent strategies employed by the LLM in managing the SafeMath library, which has traditionally been used in Solidity programming to prevent arithmetic overflow and underflow. In many instances, the language model retained SafeMath even though recent versions of Solidity now include built-in overflow checks. While it is understandable to want to maintain legacy interfaces, the decision to keep SafeMath when it is no longer necessary raises questions about the effectiveness of the upgrade. Retaining outdated dependencies can complicate the code and may introduce vulnerabilities, especially if developers are not fully aware of the implications of using such libraries in a modern context. On the other hand, contracts that have successfully removed SafeMath in favor of Solidity's built-in safety features demonstrate a positive adaptation to evolving best practices. However, the inconsistency in this approach across different contracts reveals a fragmented understanding of these advancements.

The LLM used a varied approach also to upgrade the occurrences of the `.transfer` method of a payable address. While it generally upgrades the contract from the first iteration, in adaptation to best practices, and replacing this invocation with the low-level method `.call{value: ... }` and the associated `require` for the success of this transaction, there are two instances where it retains `.transfer`, despite the fact that its use is currently not recommended for security reasons.

In some cases, legacy contract present a function to destroy the contract. To require a replacement of the deprecated `selfdestruct`, it was necessary to interact with the address the compilation warnings in the upgraded contract. To maintain the contract's destruction functionality while avoiding the use of `selfdestruct`, the language model introduces a locking mechanism that disables all features of the contract and a contextual transferring of the contract's funds to the contract owner.

Notably, the LLM typically adds an *MIT license* when none is specified, although there is one instance where it does not.

Table 5
Results of the manual inspection of upgraded contract in comparison with the legacy code.

Name	Authors Notes	Automatic Report
Alcanium	Retains and uses SafeMath, corrects interface usage, specifies returns, limits function visibility, and adds require messages.	Confirms redundancy of SafeMath.
BNIToken	Avoids SafeMath for math operations, modifies SafeMath by eliminating checks, renames abstract correctly, uses "unchecked" block to limit gas, and adds require messages.	Confirms changes.
FreePalestine	Retains SafeMath with checks and implicit return, adds require comments, keeps mapping visibility unchanged, and does not use unchecked.	Confirms redundancy of SafeMath; shows few code changes instead of descriptions; coverage: 78% (low).
NanaInu	Specifies SafeMath returns, changes visibility to external, adds require messages; identical to Alcanium.	Confirms changes.
Omosubi	Eliminates SafeMath, replaces üint256(0) with type(uint256).max.	Confirms changes.
O2OToken	Removes SafeMath checks without using it, adds constructor keyword, employs interface keyword, removes var type, specifies public, virtual, and override modifiers, and adds require messages.	Confirms that SafeMath is now unnecessary.
Shop	Retains SafeMath with syntax changes, adds require messages, removes now for block.timestamp, adds receive and fallback functions, and removes .transfer.	Confirms changes; lacks practical motivation for new SafeMath usage; shows code changes instead of descriptions; test coverage: 72% (all core functions tested).
TokenMintERC20Token	Retains SafeMath, replaces .transfer with .call, and makes minor changes to visibility and overriding keywords.	States built-in overflow checks are used but retains programmed checks.
Eloncat	Retains SafeMath, uses calldata, and adds "excludeFromReward" and "includeInReward" functions.	Confirms SafeMath is kept for identical behavior; new functions added for implied functionality.
Entropy	Removes SafeMath and now, replaces uint256(-1) with type(uint256).max, and replaces assembly with block.chainid.	Confirms changes.
PonderAirdrop	Removes SafeMath checks, corrects visibility and modes, adds require comments, and implements a kill function without selfdestruct, disabling the contract and recovering funds; emits freeze twice unnecessarily.	Confirms changes but does not explain double event.
BlindAuction	Replaces .transfer with .call, adds require messages, and removes now for block.timestamp.	Confirms changes.
SimpleAuction	Continues using .send(amount) and removes now for block.timestamp.	Confirms changes.
Ballot	No substantial code changes.	Confirms changes.
EtherTool	Uses SafeMath unchanged (asserts without messages), adds require messages, replaces .transfer with .call, and updates batchTransfer to return amounts to msg.sender if one fails; adds receive and fallback functions.	Confirms changes and describes safety upgrades.
Purchase	Retains .transfer.	Does not mention .transfer.
ReceiverPays	Replaces .transfer with .call, adds require messages, and replaces selfdestruct with a simple fund emptying.	Confirms changes.
SepukuToken	Retains SafeMath, specifies abstract contract and override functions, and adds require messages.	Confirms changes.
SimplePaymentChannel	Retains .transfer, updates now with block.timestamp, adds isChannelOpen state variable and modifier, changes selfdestruct to empty funds to the contract creator, and adds events.	Does not mention .transfer or new events.
Timelock	Converts contract to interface, updates built-in modifiers (constant -> view), adds a receive function (no fallback), and replaces .now with block.timestamp; adds require comments.	Confirms changes.
IcoLib	Retains SafeMath by modifying checks, removes direct assignment of owner = msg.sender, adds require messages, uses library style for SafeMath calls, specifies virtual without using interface, and adds constructors and receive/fallback functions; replaces assembly with high-level delegatecall, heavily modifies setVesting, adds emit on claimVesting, and improves readability.	Partially confirms changes but omits details on setVesting, new emit, and returns.

4.3. Analysis of Auto-generated Reports

The third analysis examines the generated reports. The generated reports generally comply with the explicit requests sent to the LLM via the prompt in step 2. The LLM created these reports using the markdown format and are available, unchanged, in the repository for this study.

Each report includes a title and begins with a summary of the operations performed by the LLM, accurately stating both the version of the legacy smart contract language and the target version, which is 0.8.20.

All reports are divided into sections. In every case, the first two sections are dedicated to the changelog and the comparison of feature counts present in both versions of the contract. The report then focuses on the test suite, evaluating it in terms of feature coverage and describing any necessary adjustments for the proper execution of the tests (sometimes in two sections, sometimes in one).

In most cases, the report outlines the benefits in terms of security, efficiency, and maintainability of the code achieved following the upgrade. It generally concludes with a brief summary or reflection on the work performed.

Although the structure is similar, the reports differ significantly in terms of detail and content representation. Specifically, the reports are uniform only in the initial summary and the subsequent tabular changelog. Beyond that, there is no consistency in representation, as the same topic may be presented in a narrative form with bullet points or in a tabular format. The Table 6 provides a quick overview of the number of subsections that the reports use to describe each section, highlighting the

Table 6
Number of Subsections per Topic in Auto-Generated Reports. An asterisk (*) next to the topic *Test Adaptation* indicates that this topic is covered in the section *Test Coverage*. The word YES in the last column indicates whether the topic related to improvements in terms of security, efficiency, and maintainability is present as a section or at least discussed in a structured manner.

	Changelog	Feature Count	Test coverage	Test adaption	Security-Efficiency-Maint.
Alcanium	3	3	4	*	NO
BNIToken	2	2	2	3	YES
FreePalestine	2	3	2	*	YES
Nanalnu	3	5	4	*	YES
Omosubi	2	3	2	4+2	YES
O2OToken	3	3	4	*	NO
Shop	2	3	3	*	YES
TokenMintERC20Token	2	3	2	3	YES
Eloncat	4	3	4	*	YES
Entropy	1	3	2	3	YES
PonderAirDropToken	4	4	4	*	YES
BlindAuction	2	3	2	3	YES
SimpleAuction	1	2	2	3	YES
Ballot	1	2	2	3	NO
EtherTool	2	2	3	3	YES
Purchase	1	2	3	*	YES
ReceiverPays	2	3	2	3	YES
SepukuToken	3	3	2	1	YES
SimplePaymentChannel	2	3	3	*	YES
Timelock	1	2	3	*	YES
IcoLib	2	2	3	*	YES

Table 7
Average values and standard deviations of the total number of subsections present in the reports, categorized by complexity level as defined in the Table 1.

	low complexity	medium complexity	high complexity
Average number of subsections	7,57	10	9,57
St. Dev.	1,99	2	1,72

significant differences between these reports. By analyzing the average total number of subsections for each report, grouped by the complexity of the legacy code, it is evident that reports for low-complexity code tend to be less extensive (averaging around 7.6 sections) compared to the other two cases, which are equivalent at approximately 10 sections each. These results are summarized in Table 7.

In the cases of the Shop, EtherTool, ReceivePays, SimplePaymentChannel contracts, and the FreePalestine token, the changelog section includes entire portions of modified code, allowing for direct comparison through "before" and "after" comments in the first three cases, or using git-style highlighting (with lines marked in green or red) in the latter. Notably, the SimplePaymentChannel report describes how the contract was modified to eliminate the use of selfdestruct. In other cases, the report highlights the names of functions or keywords in the code that were modified.

The impact assessment is generally presented as a structured list divided into three parts, one for each topic. In the case of SepuToken, the impact assessment includes a qualitative score for the three categories considered: Security score, Efficiency score, and Maintainability score (indicating Security as enhanced, Efficiency as neutral, and Maintainability as significantly enhanced).

Regarding the management of SafeMath, in cases where the upgrade removes or inhibits this library, the report highlights the benefits of using built-in overflow protection in terms of contract conciseness, gas usage, and security. Where SafeMath is retained, the report emphasizes how the new contract maintains the same behavior and structure as the legacy contract.

5. Discussion

This work represents a preliminary and exploratory study that examines a dataset of 21 legacy sources and a single LLM, specifically Claude 3.7. The primary objectives are to evaluate the LLM's approach, enable manual examination, and obtain both quantitative and qualitative assessments, as well as insights for a more comprehensive study that would involve a systematic and automated comparison between multiple LLMs and a wider range of smart contracts.

The ability of the LLM to produce functioning code with zero or few iterations is encouraging, as

it demonstrates the model's understanding of the characteristics and keywords of the Solidity 0.8.x language, allowing it to perform upgrades by modifying the relevant parts of the legacy code while preserving the logic and interface of the functions.

The need to iterate multiple times for the tests suggests that the LLM may not have a complete mastery of creating comprehensive test suites. There were a few cases where additional prompt details had to be provided to guide the LLM towards returning consistent responses. In other instances, minor syntax issues were manually corrected to avoid unnecessarily overloading the chat interactions.

The findings also suggest that the length and complexity of the legacy code seem to influence the number of iterations required and the quality of the generated reports.

Additionally, the fact that the LLM's proposed solutions were not coherent across all upgraded smart contracts indicates the need to further refine the prompts to establish clear rules for handling certain situations. For example, it may be beneficial to encourage code changes that align with best practices rather than maintaining the same interface, as this has led to anomalous behavior when updating `SafeMath`. Similarly, this approach should be applied to the handling of `transfer` and `selfdestruct` functions.

The stylistic differences and varying levels of detail in the generated reports suggest the need to better structure the prompts, potentially by providing guidance on the required format. Lastly, the study has identified some interesting insights, such as the inclusion of a changelog with the code and the potential for an enhancement score.

6. Conclusions

This initial study explored the potential of Claude 3.7 to automate the upgrade of legacy Solidity smart contracts to version 0.8.20. The adopted methodology allowed for a satisfactory evaluation of both the LLM's performance and the approach itself. Our findings highlight both the capabilities and limitations of using LLMs for this specialized task. Claude 3.7 demonstrated the ability to generate upgraded and compilable smart contracts with minimal iterations, particularly for low and medium complexity contracts. However, performance significantly degraded with increased contract complexity, indicating that high-complexity contracts required more iterations to achieve satisfactory results.

While the model successfully adapted core syntax, it exhibited inconsistencies in handling critical security aspects, such as the variable treatment of the SafeMath library and the updating of deprecated functions. These inconsistencies suggest that, although the LLM can automate much of the upgrade process, it lacks a consistent understanding of security best practices.

The generation of test suites proved to be more challenging for the model, requiring more iterations than the contract upgrades themselves, which indicates that creating effective test suites remains a complex task. The reports generated by the model showed a solid understanding of the upgrade dimensions, but their structure and detail varied significantly, with simpler contracts receiving slightly less extensive documentation.

This research demonstrates that Claude 3.7 has promising capabilities to reduce the manual effort required in smart contract upgrades. However, its limitations indicate that it should be viewed as an assistive tool rather than a complete replacement for human expertise.

We interpret our results in the context of smart contract security and upgrade automation, acknowledging potential biases and limitations of the study design. It is clear that human intervention remains necessary in scenarios involving complex contracts and critical security considerations. Future research should focus on expanding the dataset to include a wider variety of smart contracts, exploring the LLM's capabilities in other blockchain programming languages, and integrating specialized security analysis tools to assess vulnerabilities in upgraded contracts. Balancing automation with human oversight is crucial, particularly for security-critical blockchain applications, allowing developers to concentrate on complex logic changes and security verifications, ultimately leading to more efficient and secure smart contract upgrades.

Acknowledgments

This work was partially supported by 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. ”

Declaration on Generative AI

During the preparation of this manuscript, the authors used Duck.AI (model GPT-4o in an anonymous form) in order to: Grammar and spelling check, Paraphrase and reword. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the publication’s content. As described within this work, the authors used Claude 3.7 to perform their investigation.

References

[1] T. D. Nguyen, L. H. Pham, J. Sun, Sguard: towards fixing vulnerable smart contracts automatically, in: 2021 IEEE Symposium on Security and Privacy (SP), IEEE, 2021, pp. 1215–1229.

[2] X. Li, J. Yang, J. Chen, Y. Tang, X. Gao, Characterizing ethereum upgradable smart contracts and their security implications, in: Proceedings of the ACM Web Conference 2024, 2024, pp. 1847–1858.

[3] J. Zhao, X. Chen, G. Yang, Y. Shen, Automatic smart contract comment generation via large language models and in-context learning, Information and Software Technology 168 (2024) 107405.

[4] B. Boi, C. Esposito, S. Lee, Smart contract vulnerability detection: The role of large language model (llm), ACM SIGAPP Applied Computing Review 24 (2024) 19–29.

[5] S. Chatterjee, B. Ramamurthy, Efficacy of various large language models in generating smart contracts, in: Future of Information and Communication Conference, Springer, 2025, pp. 482–500.

[6] A. Pinna, M. I. Lunesu, S. Orrù, R. Tonelli, Investigation on self-admitted technical debt in open-source blockchain projects, Future Internet 15 (2023) 232.

[7] 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 164 (2025) 107563.

[8] M. Rodler, W. Li, G. O. Karame, L. Davi, {EVMPatch}: Timely and automated patching of ethereum smart contracts, in: 30th usenix security symposium (USENIX Security 21), 2021, pp. 1289–1306.

[9] T. Hossain, F. H. Bappy, T. S. Zaman, T. Islam, Seam: A secure automated and maintainable smart contract upgrade framework, arXiv preprint arXiv:2412.00680 (2024).

[10] F. Barbàra, E. A. Napoli, V. Gatteschi, C. Schifanella, Automatic smart contract generation through llms: When the stochastic parrot fails, in: 6th Distributed Ledger Technology Workshop, 2024.

[11] E. A. Napoli, F. Barbàra, V. Gatteschi, 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.

[12] R. Karanjai, L. Xu, W. Shi, Teaching machines to code: Smart contract translation with llms, arXiv preprint arXiv:2403.09740 (2024).

[13] E. A. Napoli, N. Romani, V. Gatteschi, C. Schifanella, Light and shadows of smart contract development with llms, Available at SSRN 5189331 (2025).

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

[15] G. Ibba, G. Baralla, G. Destefanis, Large language models for synthetic dataset generation: A case study on ethereum smart contract dos vulnerabilities (2025).