Light and shadows of smart contract development with LLMs

Expert Systems With Applications 296 (2026) 129011

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

Light and shadows of smart contract development with LLMs

Emanuele Antonio Napoli [a,*], Noemi Romani [a], Valentina Gatteschi [a], Claudio Schifanella [b]

[a] Politecnico di Torino, Corso Duca degli Abruzzi, 24, Turin, 10129, Italy
[b] Università degli studi di Torino, Via Verdi, 8, Turin, 10124, Italy

ARTICLE INFO

Keywords:
Smart contract development
Smart contract generation
Smart contract
Blockchain
Large language model
Legal contracts

ABSTRACT

Smart contract development remains almost inaccessible to non-experts developers despite the growing adoption of blockchain technology across industries. This paper evaluates the potential of Large Language Models (LLMs) for automated smart contract generation from legal agreements. The work systematically assesses the capabilities of four leading commercial LLMs – `gpt-4-turbo` (OpenAI), `claude-3.5-sonnet` (Anthropic), `mistral-large` (MistralAI), and `gemini-1.5-pro` (Google) – across a diverse range of legal agreements with varying complexity. The evaluation framework consists of a in-depth evaluation of structured code patterns – typical to smart contracts – to provide nuanced insights into model performances. The results reveal a performance hierarchy with `claude-3.5-sonnet` and `gpt-4-turbo` consistently outperforming `mistral-large` and `gemini-1.5-pro`, particularly when handling complex agreements such as mortgage note agreement and property sales agreement. A nonlinear relationship has been observed between contract complexity and model performance, with even top-performing models showing significant degradation when processing intricate legal structures. Although achieving syntactic correctness has become increasingly feasible, ensuring functional completeness and security remains challenging, as evidenced by high-impact vulnerabilities detected across all generated smart contracts. This work contributes to the growing discourse on LLM applications in blockchain technology by providing empirical evidence of current capabilities and limitations, establishing a robust foundation for future research in AI-assisted smart contract development.

1. Introduction

The convergence of blockchain technology and artificial intelligence, particularly Large Language Models (LLMs), has emerged as a promising frontier in the domain of smart contract development. As blockchain technology continues to revolutionize various sectors with its core properties, such as transparency, immutability, and reliability, the complexity of SC development remains a considerable obstacle for those without expertise in this field. Large Language Models have been tested in a wide range of applications such as medical (Healey et al., 2025; Trager et al., 2025), finance (Li et al., 2023; Liu et al., 2024), and computer science (Husein et al., 2025; Zubair et al., 2025). However, despite their widespread adoption and promising results in these domains, there remains a significant debate in the scientific community about whether the current enthusiasm around LLMs represents a technological hype cycle or a genuine paradigm shift. This is particularly evident in software development, where LLMs have shown impressive capabilities in code completion and bug fixing tasks (Husein et al., 2025). Although some developers report substantial productivity gains when using LLM-powered tools for routine coding tasks, others argue that the limitations of the model, code quality, and secure software development make them better suited as assistive tools rather than replacements for human programmers (Espinha Gasiba et al., 2024). The question of whether LLMs can truly revolutionize labor-intensive tasks like programming remains open, with empirical evidence still emerging about their real-world impact on developer productivity and code quality. Therefore, there is a pressing need to systematically evaluate LLMs' capabilities in coding tasks, particularly in domains where code reliability and security are paramount, such as smart contract development, where even minor vulnerabilities can lead to significant financial losses or security breaches.

With this objective, this paper starts from the findings of Napoli et al. (2024) and presents an extended and comprehensive study of automatic smart contract generation using state-of-the-art LLMs. In particular, this paper includes a broader range of legal agreements and evaluates the performance of four leading commercial LLM models: `GPT-4-Turbo` by OpenAI, `claude-3.5-sonnet` by Anthropic, `mistral-large`

* Corresponding author.
E-mail addresses: emanuele.napoli@polito.it (E.A. Napoli), noemi.romani@polito.it (N. Romani), valentina.gatteschi@polito.it (V. Gatteschi), claudio.schifanella@unito.it (C. Schifanella).

https://doi.org/10.1016/j.eswa.2025.129011
Received 12 March 2025; Received in revised form 5 June 2025; Accepted 11 July 2025
Available online 17 July 2025

by MistralAI, `gemini-pro` by Google. This comparative analysis aims to provide deeper insight into the capabilities and limitations of different LLM architectures in the context of smart contract creation.

This research contributes to the growing body of knowledge at the intersection of blockchain and AI in several key ways.

1. It offers a comparative analysis of smart contract generation capabilities in multiple leading LLMs, providing information on their relative strengths and weaknesses.
2. It presents an enhanced evaluation framework that combines automated metrics with manual code pattern analysis, offering a more comprehensive assessment of generated smart contracts.
3. It explores the potential of LLMs to bridge the knowledge gap for non-experts in blockchain technology, potentially democratizing access to smart contract development.
4. It investigates the ability of LLMs to accurately translate complex legal agreements into functional smart contracts, addressing a critical need in the blockchain ecosystem.

This paper aims to provide a more robust and practical foundation for the development of tools that can assist in the automatic generation of smart contracts. This research not only advances the state-of-the-art in AI-assisted smart contract creation, but also contributes to the broader goal of making blockchain technology more accessible and user-friendly for a wider audience.

The remainder of the paper is organized as follows. Section 2 reviews related works on smart contract generation, covering the main approaches which are template-based, model-driven, visual-oriented, and LLM-based. Section 3 details the proposed pipeline, the methodology followed, and the evaluation metrics used to assess the quality of the generated smart contracts. Section 4 presents a discussion of the results. Section 5 concludes the paper by summarizing key findings and outlining potential directions for future research.

2. Related work

In recent years, blockchain technology has received increasing interest from the scientific community. This emerging technology has attracted significant attention (Luu et al., 2016) due to its potential to transform numerous domains, including medicine, finance, social good, copyright, supply chain management and the legal sector. The latter could greatly benefit from the smart contract technology, which makes possible the automatization of legal clauses, including financial and ownership terms. In response, researchers and practitioners have devoted considerable effort to the creation of methodologies for the automatic and semi-automatic generation of smart contracts. The most widely used methodologies are template-based (TA), model-driven (MDA), and visual-oriented (VOA) approaches (Dixit et al., 2022). Since the release of the first commercial Large Language Model, by OpenAI, in 2022, the paradigm of smart contract generation, among others, has undergone a dramatic transformation. Using LLMs, smart contracts can now be developed simply by describing its business logic in natural language, eliminating the need for additional technical knowledge or expertise. Because of its ease of use, smart contract generation using LLMs (Large Language Models Approach, LLMA) has the potential to revolutionize the smart contract development paradigms used to date, perhaps making them obsolete. The relevant state-of-the-art works are summarized in Table 1 which presents the approaches used among the aforementioned ones (TA, MDA, VDA and LLMA) and the programming

Table 1
Landscape of smart contract generation approaches: template-based (TA), model-driven (MDA), visual-oriented (VOA), and LLM-based (LLMA). A checkmark (✓) indicates implementation, while (×) denotes absence.

Work	Methodology	Publicly Available	Programming Language	Use Case	Vulnerability Assessment
Hao et al. (2023)	TA	×	Python	Basic functions Interfaces Libraries	SmartCheck Tikhomirov et al. (2018)
Fang et al. (2023)	TA	✓	Python, JavaScript	Agreement Contract Credit Contract Employment Contract Indenture Contract Joint Filling Contract Plan of Merge Agreement Registration Right Agreement Security Purchase Agreement Trust agreement Underwritten Agreement	SmartCheck Tikhomirov et al. (2018) Securify Tsankov et al. (2018)
Zhao et al. (2023)	TA	✓	Solidity	Maritime Transport	×
Tsai et al. (2019)	MDA	×	Timed Automata	Real-estate Purchase	UPPAAL[1]
Jurgelaitis et al. (2023)	MDA	✓	UML, Go, Solidity	Hackatlon certification issuance	HFCCT Li et al. (2022) Slither Feist et al. (2019)
Köpke et al. (2023)	MDA	✓	JavaScript	Road misconstruction claims	×
Tsiounis and Kontogiannis (2023)	MDA	×	Domain Specific Language	E-Commerce	×
Franz et al. (2019)	VOA	×	JavaScript	×	×
Mao et al. (2019)	VOA	×	Python, JavaScript	fundamental function codes including transfer functions ERC20 standard interface codes SafeMath basic contract codes	×
Shen et al. (2023)	VOA	✓	JavaScript	Voting and Election Contract Voting Result Notification Contract Crowfunding Contract Supply Chain Contract Buy and Selling Contract	Slither Feist et al. (2019)
Trestioreanu et al. (2023)	VOA	✓	JavaScript	×	×
Qasse et al. (2023)	LLMA	✓	Java	Digital certificate, Medical records	iContractML verifier
Petrović and Al-Azzoni (2023)	LLMA	×	Python	Grade system Vehicle auction	×
Our	LLMA	✓	Python	Rental Agreement Property Sale Agreement Mortgage Note Agreement Equipment Rental Agreement Compensation Agreement Cleaning Service Agreement	Slither Feist et al. (2019)

language used to develop the proposed tool. It also indicates whether the approach includes a vulnerability assessment and presents a case study.

In this Section, relevant approaches of the above-mentioned automatic smart contract methodologies will be discussed, highlighting advantages and limitations of each approach.[1]

2.1. Template-based approach

The template-based approach to smart contract generation has been the most widely utilized method for this purpose. The template-based approach represents a strategic methodology for creating smart contracts and legally binding contracts by utilizing pre-designed frameworks and standardized formats. By providing a structured foundation that can be easily customized, these templates significantly reduce the time and effort traditionally associated with document creation, allowing more efficient and precise contract drafting processes. Famous commercial systems such as The AccordProject,[2] CommonAccord,[3] OpenLaw,[4] implement Richardian contracts. Proposed by Grigg (Grigg, 2004), the Ricardian contract represents an innovative approach to digitizing legal agreements by transforming them into cryptographically verifiable digital documents. This technology enables the creation of agreements that are simultaneously comprehensible to non-legal professionals through human-readable text and securely stored on blockchain, facilitated by processes of hashing and digital signing. However, these solutions do not qualify as smart contracts in the strict sense, as they are essentially document templates with fillable fields that, once completed, can be cryptographically signed and stored on the blockchain. Unlike smart contracts, they do not operate as executable scripts running on the blockchain.

Hao et al. (2023) state that the main reason for security problems in smart contracts is unnormalized code due to coding processes that are often complex. In order to mitigate this issue, the authors propose a method for the generation of smart contract's templates based on the Long Short-Term Memory Recurrent Neural Network (LSTM-RNN) to simplify the coding process. After a preliminary phase of smart contract crawling on Etherscan, a pre-processing phase in which the most appropriate and secure smart contracts are selected, the neural network is trained. The authors tested the tool on the generation of three types of base code: basic method code (e.g. *transfer*, *approve*, *transferFrom*), standard interface code (such as the ERC20 token standard) and basic contract code (such as *SafeMath* Contract Library and *StandardToken* Contract Template). The effectiveness of the model is evaluated by comparing vulnerabilities in smart contracts developed by students, both with and without the tool's assistance. Although 95.48 % of the smart contracts developed without using the generated templates contained vulnerabilities, this number decreased to 83.73 % when students used the tool. Fang et al. (2023) introduce iSyn, a semi-automatic tool designed to generate smart contracts from legal agreements. The tool employs a technology stack consisting of Node.js, Solidity, and Natural Language Processing (NLP). It has been tested on various legal agreements, including employment agreements, security purchase agreements, stock purchase agreements, registration rights agreements, and plan and merger agreements. iSyn leverages NLP to extract information from legal agreements. However, it requires manual selection of relevant data to construct the intermediate representation proposed by the authors. Additionally, iSyn generates a series of tests that the resulting smart contract must pass. While iSyn achieves commendable results, its template-based design does impose certain limitations. For example, the tool's modeling capabilities are restricted to specific actions such as payments, purchases, file uploads, and time-limited activities. However, it does not account for elements such as late fees, deposits, and price variations, which may be specified in legal agreements. These limitations underscore the challenges involved in adapting such tools for the complete generation of smart contracts. With regard to the supply chain, Zhao et al. (2023) developed MariSmart, a framework for maritime transportation based on Solidity templates, which has been released to the public. In MariSmart, the business logic is based on real-world practices and then translated into a set of Solidity templates that model the shipment and stakeholders' actions. As the solution is template-based, it can be customized with respect to parameters, activities, and workflows.

The above overview of recent work highlights that template-based smart contract generation simplifies development by reducing technical complexity, enhancing coding speed, and promoting code standardization. It also enhances security by using templates with fewer vulnerabilities and encourages code reuse, streamlining development for various business scenarios. However, while this approach may limit the flexibility of modifying a smart contract template when requesting new features, excessive customization could compromise its security and reusability advantages.

2.2. Model-driven approach

The model-driven approach (MDA), akin to the template-based approach, has been employed for the development of smart contracts. This methodology prioritizes abstract representations of the knowledge and activities specific to a given application domain, rather than focusing on computing concepts. Using strict models and transformation rules, the MDA facilitates the automation of the generation process. In this methodology, sources are assigned to targets through templates, resulting in the creation of a formal model that produces the final smart contract. Furthermore, MDA supports rapid, interactive development and promotes human-friendly communication.

In the literature, MDA-based generation of smart contracts has been investigated in several works. Tsai et al. (2019) introduce Beagle, a model-driven framework designed for smart contract development within the legal domain. Rather than replacing legal agreements, Beagle aims to partially automate the execution of legal contracts to generate legal evidence. The framework is structured into five distinct stages: template production, formal model and code development, execution, verification and validation, and runtime monitoring. The main stages are performed using UPPAAL,[5] an integrated tool environment for modeling, validation, and verification of real-time systems modeled as networks of timed automata, extended with data types. Köpke et al. (2023) propose SecBPMN2BC, a model-driven approach to design secure business processes that can be deployed as smart contracts on blockchains. The method extends BPMN 2.0 (Scheruhn et al., 2015) with security-specific annotations, includes algorithms to identify and resolve conflicts among security requirements, and provides a structured workflow to guide the design and implementation process. The approach was validated using a real-world use case involving road misconstruction claims, where processes such as claim reporting, urgency assessment, and resolution planning were modeled and implemented. The method demonstrated its ability to integrate security-by-design principles into smart contract development while leveraging blockchain features like transparency and immutability. Finally, Jurgelaitis et al. (2023) present MDAsmartSC, a model-driven methodology designed to streamline smart contract development. This approach employs model-to-model and model-to-text transformations to produce smart contract code in Go (for Hyperledger Fabric) and Solidity (for EVM-compatible blockchains, such as Ethereum). Tsiounis and Kontogiannis (Tsiounis & Kontogiannis, 2023) present a model-driven approach aimed at automating the development of smart contract systems. It employs

[1] https://uppaal.org/
[2] https://accordproject.org/
[3] http://www.commonaccord.org/
[4] https://www.openlaw.io/
[5] https://uppaal.org/

extended goal models to represent stakeholder objectives, dependencies, and policies, which are then translated into Solidity smart contract skeletons. This transformation is facilitated by a custom domain-specific language (DSL) that formalizes these models and ensures that the generated code adheres to the specifications. The method was tested using an e-commerce use case, where the process involved tasks such as login, order creation, and payment, governed by conditions and policies such as authentication and payment validation. The generated smart contracts were deployed on Ethereum, demonstrating the model's ability to enforce workflows and ensure compliance with defined requirements while handling failures using automated rollback mechanisms.

As highlighted in the above work, the Model-Driven approach simplifies smart contract development by enabling domain experts to visually design, reducing errors, and supporting code reuse across platforms like Ethereum and Hyperledger Fabric. However, challenges include limited expressiveness for advanced features, the reliance on custom tools, the significant time investment required to learn how to use the selected tool to model business logic, and the need for rigorous validation to ensure security and reliability. Further advancements in tools and methodologies are essential to fully harness MDA's potential.

2.3. Visual oriented approach

Visual oriented approach (VOA) is commonly used by individuals with moderate programming skills to generate code through graphical representations. Users typically start by visually selecting data and processes to include in the program, and then utilize program tables, tabular representation used to define programming logic and workflow, to produce the target code. A popular framework for creating such visual solutions is Google's Blockly,[6] an open source library that allows the development of web-based visual programming editors. Blockly provides user-friendly drag-and-drop building blocks, making it easier to construct programs. In the context of VOA, Franz et al. (2019) introduce a smart contract generator with a wizard-style interface that collects small pieces of information step by step, thus minimizing user overwhelm by avoiding the display of all input fields at once. The generated source files are processed by Solidity compilers to produce the corresponding binaries. However, a significant limitation of this model is the lack of an integrated smart contract deployment process within the generator environment. The prototype is built using Node.js, and future developments may include the integration of a Truffle environment, enabling the compilation and deployment of smart contracts on a blockchain. The Char-RNN model, proposed by Mao et al. (2019), presents an interactive approach to smart contract design aimed at non-programmers. The system uses Ethereum blockchain explorers, such as Etherscan and Etherchain, to crawl smart contract program datasets. Implemented in Python with the PyQuery library, this crawler extracts domain-specific characteristics using the Term Frequency-Inverse Document Frequency (TF-IDF) and K-means++ clustering algorithms (Xu et al., 2014). Within this framework, the Char-RNN model learns these features to generate unified fundamental function codes, including transfer functions, ERC20 standard interface codes, and SafeMath basic contract codes. These function codes are then integrated into Google's Blockly interactive UI controls, allowing users to design and save customized smart contracts. Shen et al. (2023) propose a framework that derives the basic functions of smart contracts from a code configuration library, based on a BPMN model. Despite efforts to streamline the development process for smart contracts across various skill levels, human intervention remains essential. This includes tasks such as understanding the proposed syntax and rules, making necessary function modifications, and adding clauses that are beyond the system's translation capabilities. Trestioreanu et al. (2023) present Blockly2Hook, a tool designed to simplify smart contract development for non-experts by leveraging established teaching methodologies, such as Blockly. The tool's effectiveness is demonstrated through a use case involving the XRP Ledger smart contract. However, a key limitation of Blockly2Hook is its lack of a vulnerability checking mechanism for the developed smart contracts.

The overview of the work just provided underscores how a visual approach to smart contract development makes it more accessible by replacing traditional code with intuitive visual blocks. This enables non-programmers to participate in the coding process and reduces syntax errors. However, this approach has limitations, including reduced expressiveness for complex tasks, scalability issues for larger projects, and reliance on external libraries that may introduce security risks. Although this approach simplifies development, rigorous testing and verification are still essential to ensure the reliability and security of the contracts.

2.4. Large language model approach

The Large Language Model Approach (LLMA) is the most recent method of generating smart contracts. Despite the limited existing research, it is emerging as the most promising method due to its versatility and user-friendliness. Users can interact with the LLMA, dynamically adding features and shaping the smart contract as required. This approach is particularly accessible to non-programmers, enabling those without advanced programming skills to obtain a decent smart contract. However, the absence of programming expertise makes it essential to thoroughly double-check the correctness of the generated contract and assess potential vulnerabilities. Qasse et al. (2023) propose a chatbot-driven approach to automatically generating smart contract source code. The chatbot interacts with users to gather critical information and populate an iContractML instance (Hamdaqa et al., 2022). This instance is subsequently used to generate the smart contract. However, this approach's DSL requires users to familiarise themselves with its terminology. For example, while iContractML refers to the members of a smart contract as "participants", users may prefer alternative terms such as "role", "struct" or "party", depending on their background and preferences. Petrović and Al-Azzoni (Petrović & Al-Azzoni, 2023) leverage OpenAI's ChatGPT to automate smart contract generation. Their framework uses iContractML as a meta-model to define the contract's business logic. A model-to-text transformation then converts the iContractML instance into a prompt that is inputted into ChatGPT. This approach allows the prompt to be customised, enabling the specification of the target Solidity version or alternative languages such as the Digital Asset Modelling Language (DAML) (Kfir & Fournier, 2019) for contract generation. Although manual verification ensures that the generated smart contracts are compilable, no dedicated tools are used for security verification. The work described in Napoli et al. (2024) presents a preliminary evaluation of smart contract generation using ChatGPT. The implemented pipeline begins with a legal agreement, uses ChatGPT to generate a corresponding smart contract and then assesses its vulnerabilities using Slither. The proposed solution was tested using a rental agreement case study due to its balanced complexity and alignment with fundamental registry purposes for which blockchain is commonly utilised. However, the authors only consider a simple legal agreement – a rental contract – and analyse the generated smart contracts of just one large language model, without comparing the results with those obtained using potential alternatives. Baralla et al. (2024) evaluate GitHub Copilot's effectiveness in Solidity smart contract development, focusing on its ability to generate code, detect vulnerabilities, fix bugs, and create unit tests. Their findings show that Copilot is effective at generating basic token contracts (ERC20, ERC721 and ERC1155) while adhering to Ethereum standards and reliably verifying contract initialisation. However, Copilot struggles with complex logic, advanced security patterns and comprehensive error handling. It often requires iterative prompts to address vulnerabilities such as reentrancy and denial-of-service attacks. While Copilot shows promise in automating certain aspects of blockchain development, its limitations in producing secure and fully functional code, especially for intricate use cases, underscore the necessity of human oversight. The

[6] https://developers.google.com/blockly

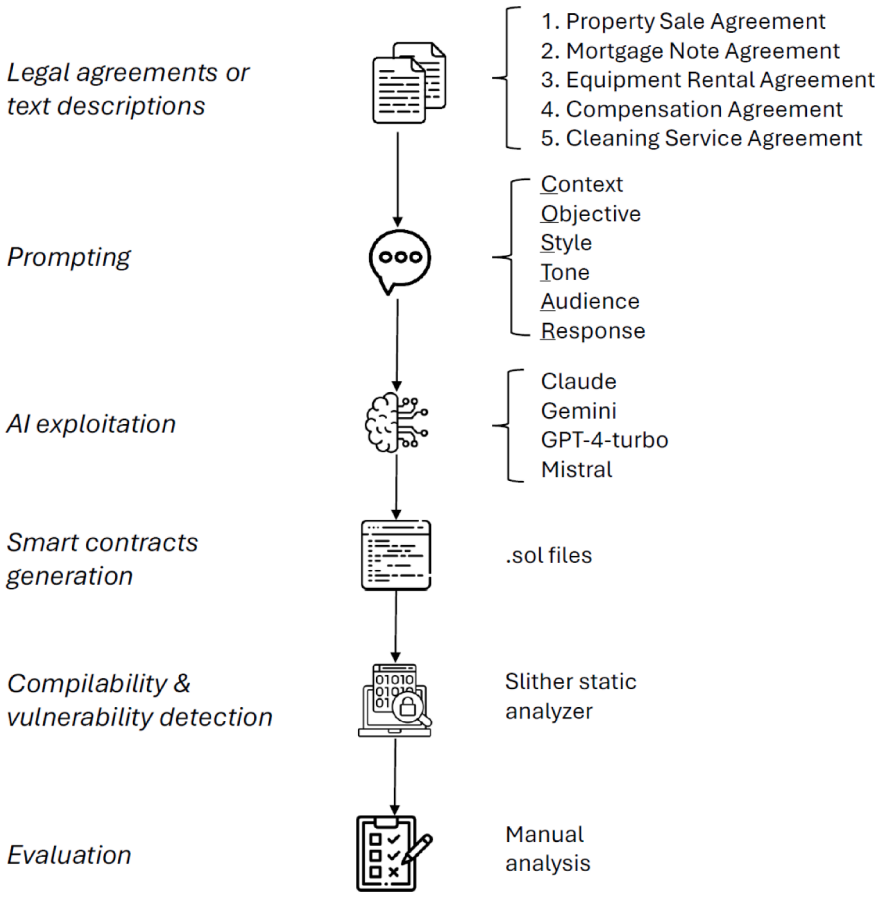

Fig. 1. Proposed pipeline.

study emphasises Copilot's utility for basic tasks, while also highlighting its challenges in ensuring robust and secure smart contracts.

While Large Language Models (LLMs) have demonstrated potential in the domain of smart contract generation, they are currently dependent on substantial human oversight and lack the reliability required for utilization as autonomous code generation tools, particularly in complex contractual scenarios that necessitate intricate logical and security considerations.

The present work constitutes a pioneering exploration in the utilization of Large Language Models for the generation of smart contracts directly from legal agreements, a novel approach that is not yet documented in the extant literature. It is acknowledged that the present research is in its infancy; however, it is believed to be the first study to systematically assess and compare the performance of LLMs across a range of types of legal agreement. This extends the scope of existing studies that have primarily focused on isolated contract generation or a single LLM. This research provides a foundational framework for future investigations into automated smart contract development, highlighting both the potential and current limitations of LLM-driven contract translation methodologies.

3. Methodology

This work extends the preliminary evaluation performed in Napoli et al. (2024), to include a thorough comparison of several LLMs, exploited for the creation of smart contracts starting from contracts belonging to different domains. Hence, in this work we compare LLMs provided by OpenAI, Mistral, Anthropic, and Google. These correspond to the respective models: `gpt-4-turbo` (GPT-4-Turbo), `mistral-large-latest` (Mistral), `claude-3-5- sonnet-latest` (Claude), and `gemini-1.5-pro` (Gemini), while retaining support for the OpenAI model.

For the comparison, we exploited the pipeline reported in Fig. 1, which operates through a structured sequence of five steps: (1) the user initiates the translation of the natural language document into a smart contract, (2) the pipeline formulates 17 distinct prompts based on the CO-STAR framework, incorporating key dimensions such as Context, Objective, Style, Tone, Audience, and Response, (3) the consolidated prompt is sent to the designated LLM endpoint for processing, (4) the generated smart contract is saved as a text file (.sol file), and (5) the integrity and security of the generated smart contract are assessed using Slither, a static analysis tool for smart contracts (Feist et al., 2019). In addition to automatic vulnerability detection performed by Slither, the generated smart contracts are manually inspected and evaluated (6). The pipeline is used to generate the smart contracts at the end of September 2024, while requirements definition and manual evaluation were performed from October to December 2024.

In order to perform the analysis on a wider range of legal agreements that could benefit the most from the usage of blockchain technology (with respect to the analysis carried out in Napoli et al., 2024), the evaluation is carried out on five legal agreements from PandaDoc,[7] named:

1. Property Sale Agreement[8]
2. Mortgage Note Agreement[9]
3. Equipment Rental Agreement[10]
4. Compensation Agreement[11]

[7] https://www.pandadoc.com/agreement-templates/
[8] https://www.pandadoc.com/kentucky-real-estate-purchase-agreement-template/
[9] https://www.pandadoc.com/mortgage-note-template/
[10] https://www.pandadoc.com/equipment-rental-agreement-template/
[11] https://www.pandadoc.com/compensation-agreement-template/

5. Cleaning Service Agreement[12]

We chose these contracts because they represent ideal candidates for blockchain integration due to their inherent characteristics of requiring immutable record keeping, transparent transaction trails, and verifiable authenticity. For example, property sales could leverage blockchain to create transparent ownership transfer records, mortgage notes could implement automatic verification of payment schedules, equipment rental agreements could track asset condition and usage in real-time, compensation agreements could ensure immediate and verifiable fund transfers, and cleaning service agreements could generate immutable performance and payment logs, thus significantly reducing dispute potential and administrative overhead while enhancing trust and operational efficiency across these diverse contractual domains.

As in Napoli et al. (2024), 6 out of the 17 identified prompts built using the CO-STAR methodology – reported in Appendix A – are used for evaluation for cost reasons. The full list of prompts can be found in the Github repository.[13] In fact, since the pipeline setting consists in running the same prompt four times, as proposed by Sobania et al. (2023) to analyze the consistency of the results, the overall number of calls to the endpoint would have been quite expensive. The total number of endpoint calls is 6 prompts per 5 legal agreements per 4 times per 4 Large Language Models, giving a total of 480 calls. A critical reflection reported in Barbàra et al. (2024) revealed that the metrics initially proposed in Napoli et al. (2024) to evaluate the generated smart contracts were potentially misleading and did not adequately reflect the true correctness of the business logic in the contracts. In fact, the holistic analysis carried out in Barbàra et al. (2024) reveals multiple flaws in the process of generating smart contracts. The analysis identified several key issues, including the absence of the body in the functions declared within the smart contract, the inability to withdraw funds from the generated smart contract, and a fundamental misinterpretation of currency.

Given these motivations, a rigorous smart contract implementation necessitates not only syntactic compilation and security validation, but a holistic assessment that accurately captures and executes the commercial requirements, encompassing comprehensive workflow dynamics, operational constraints, and transactional interactions. To address this limitation, we adopted a more nuanced and comprehensive evaluation approach. Drawing inspiration from the seminal works of Bartoletti and Pompianu (Bartoletti & Pompianu, 2017) and (Khan et al., 2023), we have implemented a manual evaluation process that examines specific design patterns identified in their research. These patterns include *Token*, *Authentication*, *Oracle*, *Math*, *Randomness*, *Pool*, *Termination*, *Time Constraint*, *Fork Check Helper*, and *Helper*.

The patterns can be briefly summarized as follows.

- *Token* pattern is used to represent the presence of transferable and countable digital or physical assets like coins, shares, or tickets, often used for tracking ownership;
- *Authentication* patterns restrict code execution by checking caller addresses, typically limiting critical operations to contract owners;
- *Oracle* pattern indicates the presence of an interfaces between smart contracts and external data sources, allowing contracts to access off-chain information while maintaining determinism;
- *Math* patterns implement logic for safe numerical operations;
- *Randomness* patterns handle the generation of random numbers, either through oracles or by using blockchain-based values, though security remains a challenge;
- *Pool* patterns enable voting functionality within contracts, often using tokens or address verification;
- *Termination* patterns provide ways to disable contracts since they cannot be deleted from the blockchain;
- *Fork check* patterns help contracts determine whether they're running on the main chain or a forked version of the blockchain;
- *Time Constraint* patterns specify when certain actions are permitted;
- *Helper* patterns serve as wrapper functions around existing functionality and include smart contract-specific operations like variable initialization (i.e. structures) and migration, essentially covering utility functions that don't fit into other categories;

Additionally, we have introduced a new pattern category, *Business Logic*, to assess the accuracy and completeness of the contract's core functionality as it relates to the intended business rules and processes.

This refined evaluation methodology facilitates a more in-depth examination of the qualitative characteristics of the generated smart contracts. The process under discussion enables the isolation of results that are not tied to a specific legal agreement. This, in turn, facilitates an assessment that goes beyond the preliminary metrics introduced in Napoli et al. (2024). This approach facilitates a comprehensive evaluation of the structural soundness of the contracts, security concerns, and alignment with the intended business logic.

As a preliminary study, some key criteria are defined for each of the patterns considered in the evaluation after several joint sessions with experts in the blockchain domain. These specifications are carefully tailored to align with the functional and structural needs outlined in the research, ensuring consistency and relevance among the legal agreements examined. Concerning the definition of the criteria, it is worth highlighting that, especially for the Business Logic pattern, the criteria were defined by experts during several sessions, in which they carefully analyzed the functionalities required by each smart contract derived from a legal agreement. Following this initial phase, each expert was randomly assigned a subset of smart contracts for evaluation. Given the large number of contracts to be assessed, it was not feasible for every expert to evaluate all contracts independently. As such, the workload was evenly distributed among the raters. To ensure consistency in scoring and minimize potential rater bias, calibration sessions were conducted for each legal agreement. Specifically, the raters participated in two preliminary calibration sessions that involved sample legal agreements and associated smart contracts. These sessions aimed to align the understanding and application of the scoring criteria of the raters. During the calibration process, discrepancies in scoring were closely examined. In cases where differences exceeded one point on the five-point scale, the raters engaged in moderated discussions to clarify their reasoning and reach a consensus on the appropriate score. This provides a robust framework for evaluating the structural and functional soundness of smart contracts. For each legal agreement, the requirements encapsulate key functionalities such as payment mechanisms, role-based access control, termination conditions, and auxiliary support structures. These requirements and their application to the various patterns are comprehensively documented in Tables B.5 and B.6 of the Appendix B.

Once the requirements are defined, blockchain experts conducted a manual evaluation by double-checking the smart contracts one by one. The process designed for the manual evaluation is the following. First of all, the manual evaluation is conducted only on the compilable smart contracts, otherwise the score of all patterns is considered null. The patterns are evaluated with a score of 0 to 4 as reported in Table 2 according to the number of satisfied requirements. Taking the "Authorization" pattern as an example, for the requirement "*The smart contract requires*

Table 2
Pattern scoring conditions.

Score	Condition
0	None of the requirements are met
1	Less than half of the requirements are met
2	Half of the requirements are met
3	More than half of the requirements are met
4	All requirements are met

[12] https://www.pandadoc.com/cleaning-service-contract-template/
[13] https://github.com/BChain4all/pipeline

that the main functions can be called only by the right party", developers counted how many functions correctly implemented access control. They calculated the number of functions that implement proper access control out of the total number of functions implemented in the smart contract. Based on this count, they assigned a score according to the criteria in Table 2.

The Slither vulnerability report is used as a reference during this manual evaluation process in order to assess the compilability of the generated smart contract. Moreover, it is used to obtain an overview of vulnerabilities in smart contracts generated from the selected LLMs, detailed in Section 4.

Once the generated smart contracts are evaluated, the score for each pattern is aggregated across $n = 4$ runs of identical prompts.

Let x_i represent the evaluation score for the run i, where $i \in \{1, 2, 3, 4\}$. The aggregate score (S) of each pattern is computed as the floor function of the arithmetic mean:

$$S = \left\lfloor \frac{1}{4} \sum_{i=1}^{4} x_i \right\rfloor$$

where $\lfloor x \rfloor$ denotes the floor function, which maps x to the largest integer not greater than x (e.g., $\lfloor 2.7 \rfloor = 2$).

This approach was chosen deliberately, as large language models can generate plausible text responses for code generation, but inaccurate responses pose significant risks in the development of smart contracts. Given the complexity of smart contracts, a conservative evaluation approach was adopted, particularly with regard to security and call costs. The evaluation method is designed to highlight potential issues rather than overstate performance. When even a single negative evaluation is present, the conservative scoring approach is applied, as it is safer to assume that there might be flaws in the smart contract that users should be aware of. This methodology prioritizes minimizing the risk of deploying poor-quality smart contracts over rewarding occasional instances of better performance, ensuring that the readers do not place excessive confidence in the results obtained through the proposed evaluation method.

The synthesis of the aggregated datasets facilitated the identification of several key findings on the effectiveness of smart contract generation using Large Language Models, which are described in the following Section.

4. Results

The results obtained are presented in Figs. 2–6 and discussed in the following, by focusing on *Overall Performance*, *Agreement Analysis*, *Prompt Analysis*, and *Vulnerability Analysis*.

4.1. Overall performance

The visualization depicted in Figs. 2 and 3 employs stacked bar charts to represent performance in smart contract generation of the selected LLMs – alphabetically ordered – across five distinct legal agreements: *Cleaning Service*, *Compensation*, *Equipment Rental*, *Mortgage*, and *Property Sale*, using five different prompts, from PR12 to PR17 already used in a previous work (Napoli et al., 2024) and detailed in Appendix A. Fig. 2 shows the results for Claude and Gemini while Fig. 3 shows the results relative to GPT-4-Turbo and Mistral. Each bar is decomposed into colored segments representing various smart contract patterns contribution, including Authorization, Fork Check Helper, Oracle, Pool, Randomness, Termination, Time Constraint, Token, Business Logic, Helper, and Math functionalities. Hence, the total score obtained is the sum of each contribution. The percentage values on each bar indicate the compilability rate of the generated contracts. This representation enables direct comparison of each model's proficiency in generating compilable smart contracts across different use cases, with notable variations in performance observed among the models. The assessment of LLMs performance across different legal agreements and prompts reveals different patterns and capabilities that are crucial for understanding their practical application for the smart contract generation task.

The `claude-3-5-sonnet-latest` model by Anthropic demonstrates remarkable consistency and superior performance in a range of tested scenarios. In particular, it achieves a 100 % rate of compilability in smart contracts generated for simpler agreements such as *Compensation* and *Cleaning Service*. This superiority is particularly evident in its handling of sophisticated patterns, including Authorization, Termination, and Business Logic implementation. The model's performance could be attributed to its robust training and subsequent reinforcement learning phase in understanding and processing legal terminology, structural requirements, and Solidity programming logic. However, its performance declines with more complex legal agreements. For instance,

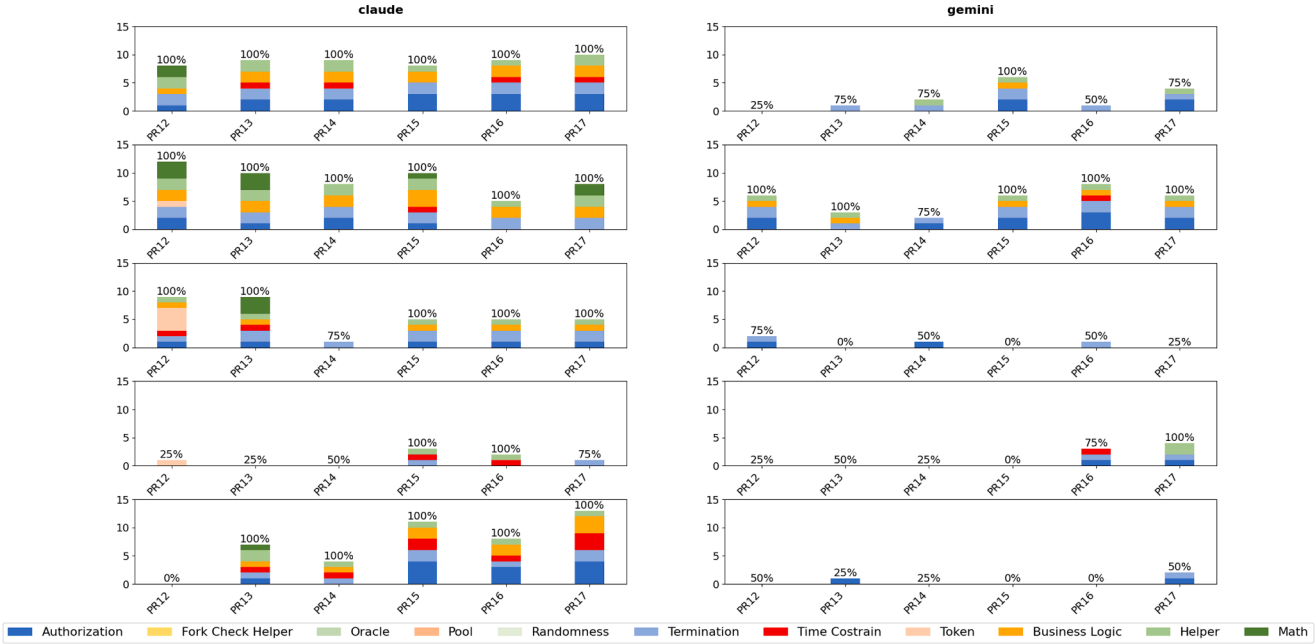

Fig. 2. Aggregated results obtained by averaging the score of each pattern of smart contracts generated by Claude and Gemini.

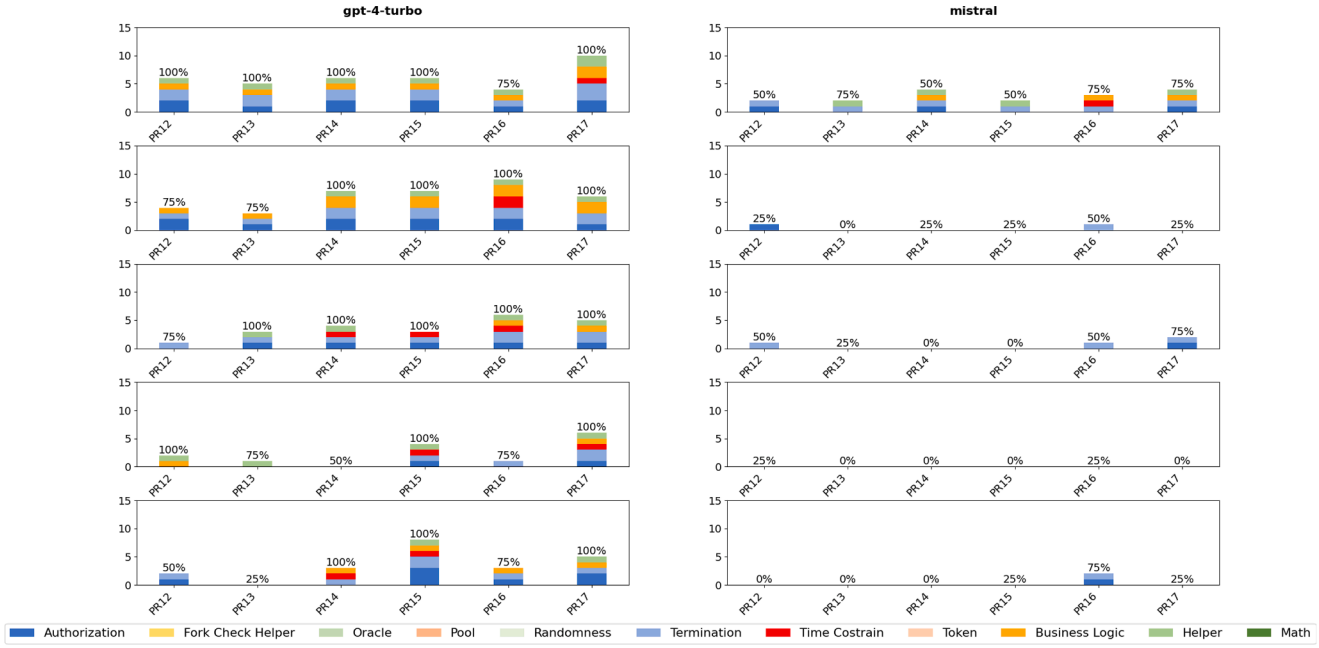

Fig. 3. Aggregated results obtained by averaging the score of each pattern of smart contracts generated by GPT-4-Turbo and Mistral.

Claude achieves a 75–100 % compilability rate for the *Equipment Rental* agreement and a significantly lower 25–100 % for the *Mortgage* agreement. In particular, it does not generate a compilable smart contract for the PR12 prompt within the *Property Sale* agreement, as shown in Fig. 2. These results underscore the limitations of the model in delivering effective smart contracts for complex legal agreements.

OpenAI's `GPT-4-Turbo` model emerges as a strong second contender, demonstrating particularly robust results in generating smart contracts for agreements such as *Cleaning Service* and *Compensation* agreements. Although it does not achieve Claude's consistent success rate 100 %, the model consistently achieves reliable compilability rates of 75–100 % in most test scenarios, as shown in Fig. 3. The model excels in handling Termination and Business Logic implementation; however, it exhibits occasional challenges with more complex Authorization requirements in *Mortgage* and *Equipment Rental* agreements. Furthermore, it struggles with the intricate Business Logic in *Property Sale* agreement. Notably, despite these limitations, `GPT-4-Turbo` consistently delivers compilable smart contracts, even when it does not meet the majority of requirements.

Google's `gemini-1.5-pro` model demonstrates moderate but inconsistent performance across various types of agreement. Its success rates generally range between 50–75 %, with occasional peaks reaching 100 % in specific scenarios, particularly for *Compensation* agreements. However, its performance declines significantly when faced with more complex agreements, such as *Equipment Rental* and *Mortgage* agreements. The model shows relative strength in handling Authorization and Termination patterns, though its results remain inferior to those of Anthropic's and OpenAI's models. Moreover, it struggles considerably with Business Logic patterns, frequently failing to satisfy even half of the required conditions. This variability indicates that Gemini could be better suited for simpler, standardized legal documents rather than complex, multi-faceted agreements.

Mistral's `mistral-large-latest` model consistently demonstrates the lowest performance metrics among the four evaluated LLMs, with success rates rarely exceeding 75 % and frequently falling below 50 %. Its performance is particularly poor in handling complex agreements such as *Property Sales* and *Mortgages*, where success rates often drop to a range of 0–25 %. Nonetheless, the model exhibits relatively better results in *Cleaning Service* agreements, indicating its potential suitability for simpler, more straightforward legal documents characterized by clear and standardized structures. However, even in these scenarios, Mistral's scores across key patterns, such as Authorization and Business Logic, are predominantly zero, further emphasizing its limitations.

This performance hierarchy (Claude > GPT-4-Turbo > Gemini > Mistral) remains largely consistent across different prompts and agreement types, although the gap between models varies depending on the complexity of the task. The difference in performance is most pronounced in complex agreements with multiple components and interdependencies, while it narrows somewhat in simpler, more straightforward document types. This pattern suggests that, while all models can handle basic legal document processing to some degree, their capabilities diverge significantly when faced with the generation of smart contract from more complex legal structures and requirements.

4.2. Agreement type analysis

From the agreement type point of view, the data reveal how different LLMs handle the smart contract generation task from various legal document structures, with each agreement type presenting unique challenges and success rates.

The *Cleaning Service* agreement consistently emerges as the type document that is processed the most successfully between all evaluated LLMs. This agreement is characterized by its clear and concise sentences, minimal clauses, and a linear organizational structure. Furthermore, as shown in Table 3, it is the second-lowest in terms of token count, making it less demanding for LLM processing. Claude achieves perfect

Table 3
Legal agreements word and average token counter assuming 1 token ≈ 4 characters on average.

Legal Agreement	Word Count	Average Token Number
Cleaning Service	859	1573
Compensation	550	1019
Equipment Rental	1305	2042
Mortgage	1239	1958
Property Sale	1140	2019

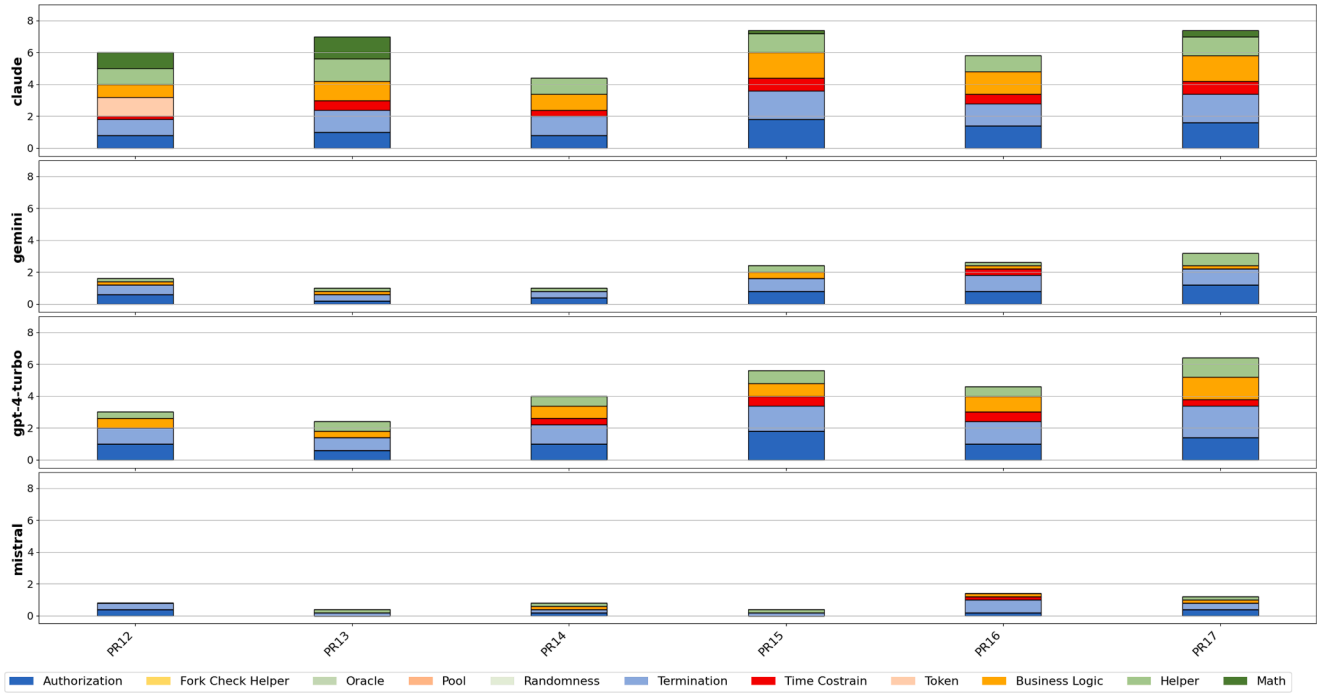

Fig. 4. Aggregated pattern results for LLM used.

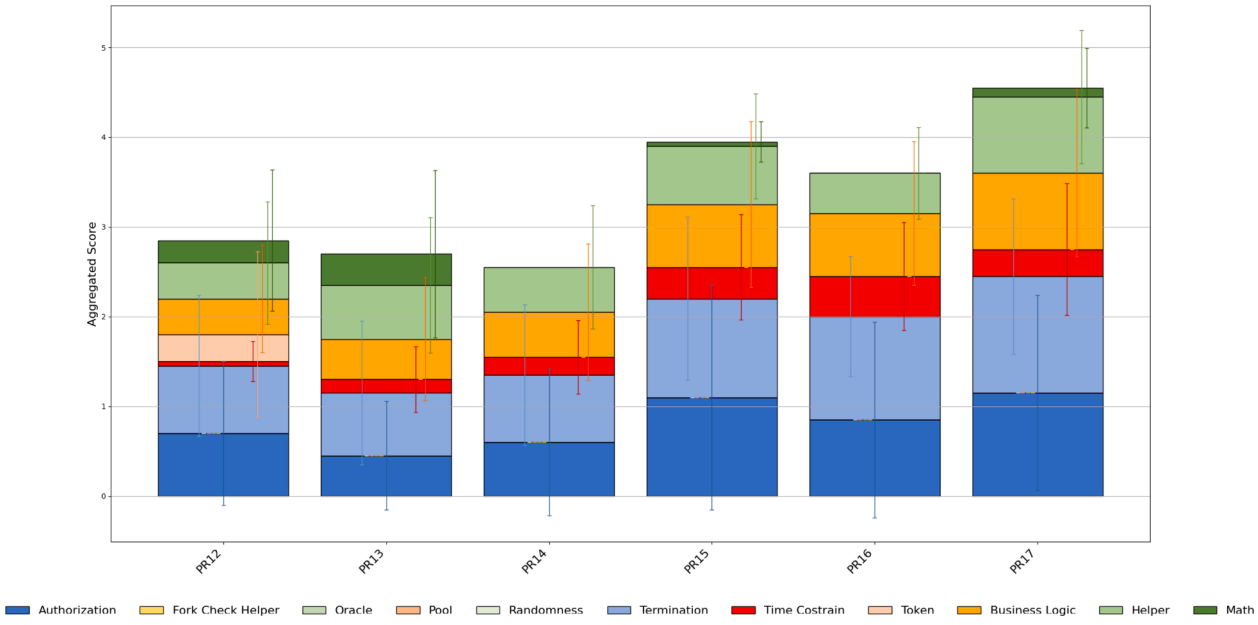

Fig. 5. Variation of pattern score for each prompt.

compilability scores (100 %) across all prompts, while GPT-4-Turbo maintains strong performance with success rates ranging from 75–100 % as evidenced in Figs. 2 and 3, respectively. Even models with typically lower performance, such as Gemini and Mistral, achieve their highest success rates in these agreements, generally ranging on average from 50–75 % as depicted in Fig. 3. This superior performance can be attributed to the straightforward nature of cleaning service contracts, their standardized structure, and the clarity of their terminology. Additionally, the high success rates suggest that these agreements feature relatively simple Authorization and Termination requirements, rendering them more accessible to all LLMs.

Compensation agreement represents an intermediate case in terms of complexity and success rates among the evaluated models. Claude maintains its exceptional performance with consistent 100 % success rates, while GPT-4-Turbo delivers strong results, achieving success rates between 75–100 %. Gemini also performs relatively well in this category, with success rates ranging from 75–100 %. However, Mistral shows a marked decrease in performance, with success rates frequently falling to 25–50 % and often not meeting any requirements. This performance pattern suggests that *Compensation* agreement demands more advanced capabilities in translating legal language into Solidity programming. The observed limitations of Google's and MistralAI's models could stem from their comparatively lower levels of training on both legal and smart contract data or their reinforcement learning through human feedback (RLHF) phase may have had minimal focus on smart contract generation and legal document processing, as these are specialized domains

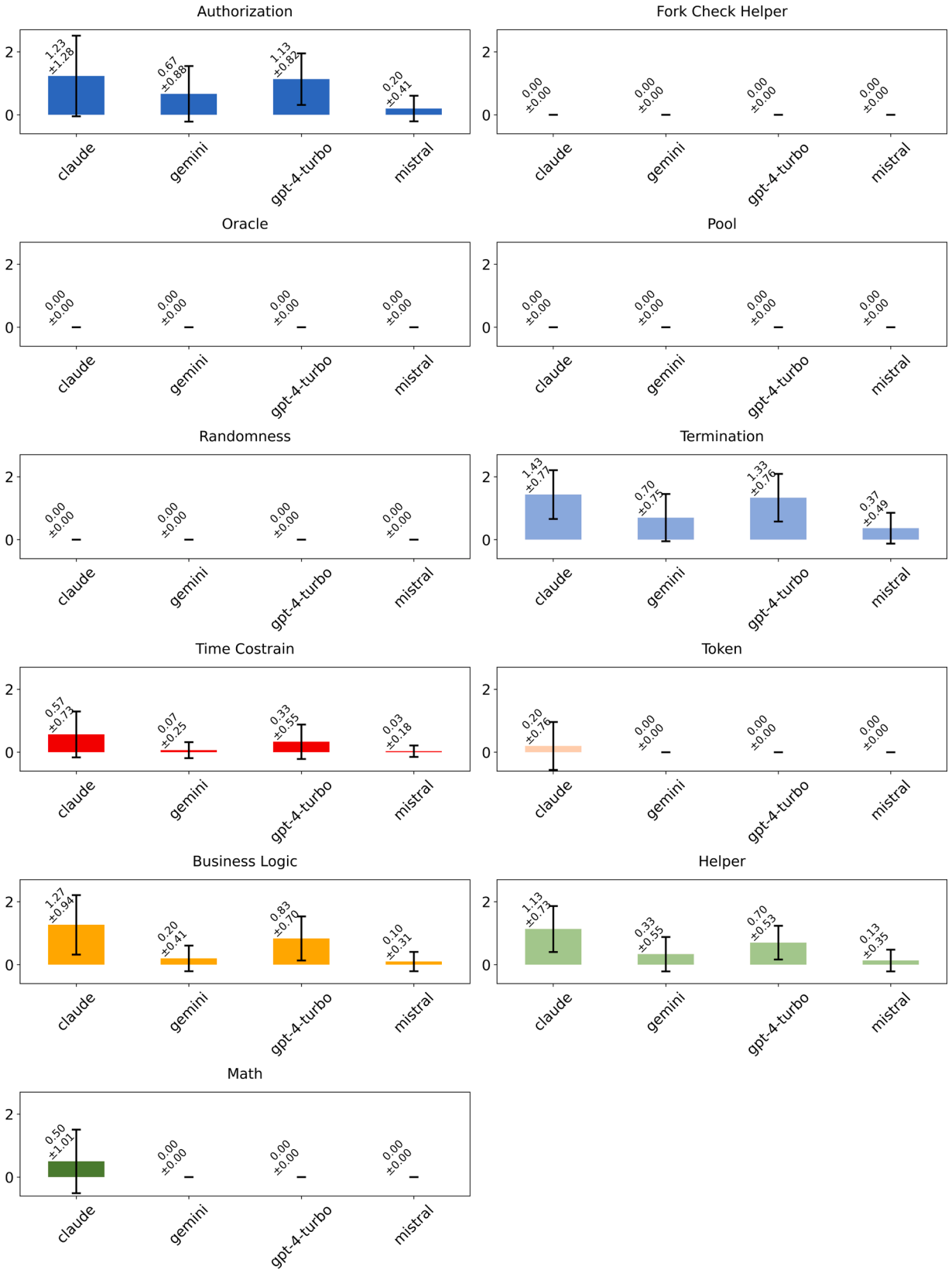

Fig. 6. Variation of pattern score for each LLM used.

requiring expert validation. This limitation becomes evident in the remaining agreements considered.

Equipment Rental agreement exhibits considerable variability in performance across the evaluated LLMs. While Claude achieves relatively high success rates (75–100 %), other models show significant fluctuations. GPT-4-Turbo performs reasonably well, with success rates also ranging from 75–100 %. In contrast, Gemini and Mistral encounter substantial challenges, often achieving success rates of only 25–50 %. This variability could be attributed to the inherent complexity of equipment rental agreements, which typically involve detailed equipment specifications, maintenance requirements, and liability clauses. Furthermore, as indicated in Table 3, the *Equipment Rental* agreement contains the largest token count among the evaluated documents. This higher token count likely places additional computational demands on LLMs, potentially affecting their reasoning and processing performance.

Property Sale agreement shows some of the lowest success rates among the LLMs evaluated. Although Claude demonstrates exceptional performance with consistent 100 % success rates in most of the prompts, other models face significant challenges. GPT-4-Turbo achieves moderate success, with rates ranging from 50–75 %, while Gemini and Mistral frequently fail to exceed 25 % success rates. This disparity underscores the complexity of property sale agreements, which often involve multiple stakeholders, complex conditions, and extensive legal requirements. Furthermore, as indicated in Table 3, Property Sale agreements rank second in token count. This higher token volume likely contributes to the challenges observed in generating compilable smart contracts for these agreements.

Mortgage agreement consistently shows the most dramatic performance variance among all types of agreement. Even Claude, the model with the highest performance, shows unusually low success rates, ranging from 25–50 % on several prompts. Other models perform even worse, with Mistral frequently recording a success rate of 0 %. In contrast, GPT-4-Turbo outperforms the other models, achieving success rates between 75–100 %. This overall poor performance can be attributed to the inherent complexity of mortgage agreements, which involve intricate financial calculations, strict regulatory requirements, and complex conditional clauses. Furthermore, as noted in Table 3, mortgage agreements rank among the highest in token count, placing significant computational demand and reasoning on LLMs and likely contributing to their diminished performance.

It can be observed that the impact of the complexity of legal agreement has a non-linear effect on the generation of smart contracts. This effect stems from the unavoidable ambiguity in natural language legal documents, which encompasses five key types: lexical (words with multiple meanings), syntactic (phrases with different grammatical interpretations), antecedent (unclear references), temporal (vague time indications), and contract reference ambiguity (imprecise terms like “hereunder” or “reasonable”). These ambiguities, while sometimes intentionally incorporated by lawyers for flexibility, frequently lead to conflicting interpretations and disputes between parties (Upadhyay et al., 2021). The intrinsic ambiguity and complex jargon make comparative quantitative analysis of legal agreements challenging. For these reasons, contract analysis presents unique challenges also for machine learning due to interconnected structures, non-standardization, and hierarchical logic that would require substantial time and resources for artificial intelligence to be processed effectively (Greenberg et al., 2024).

Despite these limitations, Balaji et al. (2024) investigated how LLMs can be used to completely automate the conversion of a legal contract to a smart contract. Their approach involves feeding `gpt-3.5-turbo` with legal agreements and applying chain-of-thought prompting techniques to extract transactions from the document. The LLM was then asked to generate possible interpretations reflecting different ways transactions could occur or be interpreted by humans, based on ambiguities in the legal language. The researchers selected the highest scoring interpretation according to acceptability and enforceability metrics before converting it into a smart contract.

Their work demonstrates that LLMs develop their own interpretations of legal agreements, probably because they are trained on a massive corpus of natural language documents where ambiguity is inherent. This helps explain the non-linear effect observed in the presented work: the LLM may produce different interpretations each time the same legal agreement is processed, due to the underlying ambiguities in the document, and for these reasons developing a formal definition of this relationship remains challenging.

Another factor that could influence the outcome of LLMs is the model parameters such as the temperature parameter. This may be considered an indication of the “creativity” of the LLM, with the potential to influence its response. However, many studies that tried to assess the effect of temperature on code generation assert that it has a negligible effect on the correctness or performance of general code generation (Coignion et al., 2024; Renze, 2024) but also of smart contracts (Napoli et al., 2024). This poor correlation of the results with temperature led to its exclusion as a factor in the experiment design of the presented work. In conclusion, since the temperature parameter does not significantly impact the code generation, the non-linear relationship observed could be primarily due to the inherent ambiguities in legal documents rather than variability in model parameters.

4.3. Prompt type analysis

The analysis of prompt effectiveness in legal document processing reveals significant patterns in how different prompts influence the performance of Language Models (LLMs) across various agreement types. As mentioned previously, this examination focuses on six distinct prompts (PR12-PR17) reported in Appendix A and their impact on document processing success rates, with particular attention to performance variations across different LLM implementations. In summary, prompts designed for an expert audience are prioritized, hypothesizing that those targeting more knowledgeable individuals would yield better smart contract generation results. For instance, while some prompts emulate a junior developer, others aim to replicate the inquiry of a senior Solidity developer.

Fig. 4 shows the performance comparison of Large Language Models evaluated on six distinct prompts (PR12–PR17) based on the data aggregation of the legal agreements under consideration. Each bar is composed of colored segments representing various smart contract patterns, including Authorization, Fork Check Helper, Oracle, Pool, Randomness, Termination, Time Constraint, Token, Business Logic, Helper, and Math functionalities. On the other hand, Fig. 5 depicts aggregated data for both LLMs and legal agreements. It presents aggregated metric averages with error bars that indicate the standard deviation. The stacked bars reveal a generally increasing trend in component implementation from PR12 to PR17, with Authorization and Termination (dark and light blue) forming consistent base components. Notable improvements appear in Business Logic (orange) and Time Constraint (red) implementations in later prompts, particularly PR15-PR17. Error bars suggest varying reliability across different LLMs and legal agreements, with wider deviations in more complex functionalities like Math and Helper components.

As anticipated above, the data in Fig. 5 demonstrates a clear evolution in prompt effectiveness, with later iterations (PR15-PR17) often showing markedly improved performance metrics compared to earlier versions. The scores aggregated by model shown in Fig. 4 reveal significant variations in this improvement. While Claude consistently maintains high performance levels (achieving higher scores on aggregated data across multiple types agreement, on average), other models such as GPT-4-Turbo, Gemini, and Mistral show more variable responses to prompt evolution. As demonstrated in Fig. 5, prompt PR17 achieves the highest score, on average, among the relevant legal agreements.

A primary contributing factor to PR17’s effectiveness is its enhanced handling of Authorization parameter, consistently meeting more of the half requirements or all requirements across different agreement types, as also Fig. 4 points out. This improvement is particularly evident in

documents such as *Cleaning Service* and *Property Sale* agreements referring to Figs. 2 and 3. Moreover, Fig. 6 shows that these score enhancements are most pronounced in Claude and GPT-4-Turbo, while Gemini and Mistral demonstrate more modest improvements in Authorization handling.

Termination handling, depicted in light blue in Fig. 4, represents another significant advancement in the design of PR17. The data indicates improved performance in temporal parameter management, with consistent scores of 2-3 (meeting half or more than half of the requirements, respectively) across various agreement types. This enhancement is particularly notable in Cleaning Service, Compensation and Property Sale agreements, where temporal dependencies often play crucial roles in document validity. Fig. 6 reveals that while all models show improvement in this area, Claude and GPT-4-Turbo demonstrate superior capability in handling complex temporal parameters.

The implementation of Business Logic also shows substantial improvement in PR17, with higher success rates in complex logical structure processing. The model-specific analysis in Fig. 6 reveals interesting patterns: Claude maintains consistently high performance in Business Logic implementation across all types of agreement, while other models show more variability, particularly in complex agreements like *Property Sales* agreement and *Mortgage* agreement. An interesting pattern emerges in Fig. 4 when examining the prompt's performance across different LLMs by aggregating data from legal agreements. While Claude maintains consistently high performance across all prompts, the score gap between different LLMs is slightly reduced with PR16 and PR17, suggesting that these prompts provide more universally interpretable instructions. However, a marked variation in performance is evident among the PR16 and PR17 models according to the nature of the agreement. The performance of the simpler agreements, such as the Cleaning Services model, exhibits greater uniformity in terms of prompt improvement across models. In contrast, more complex agreements persist to demonstrate substantial performance disparities between models, as illustrated in Figs. 2 and 3. Fig. 5 also reveals an interesting correlation between prompt complexity and success rates. PR15 achieves an optimal balance between instruction specificity and clarity, avoiding the potential pitfalls of both over-specification (as seen in some aspects of PR16) and under-specification (evident in earlier prompts). TThis balance is of particular importance in the context of handling edge cases and exceptional conditions in document processing. In fact, PR15-PR16 consistently generate a compilable smart contract in 75–100 % of the cases, with the exception of MistralAI and three legal agreements for Gemini. However, it is important to note certain limitations in the prompt's effectiveness. Performance metrics for *Mortgage agreements* remain relatively low across all prompts and models, with even Claude showing reduced performance compared to other types of agreements. This consistent challenge across all models suggests that some document types may require specialized prompt structures, regardless of the underlying model capabilities.

The analysis highlights significant variations in prompt effectiveness, depending both on the complexity of the agreement type and on the capabilities of the underlying model. Simpler agreements, such as *Cleaning Services*, consistently yield high success rates across multiple prompts and models. Conversely, more intricate agreements, such as mortgages, demonstrate greater variability in prompt effectiveness and more pronounced performance discrepancies between models. These findings suggest that future prompt design iterations should adopt a more granular and document-specific approach and consider the capabilities and limitations of the model.

It is important to note that this study employs a one-shot methodology to generate smart contracts directly from legal agreements. This approach is chosen to simulate the experience of an average, non-expert user with no background in legal or programming domains. Although the one-shot method aligns with the intended user scenario, we strongly believe that adopting alternative methodologies, such as chain-of-thought prompting techniques, could significantly enhance the performance of the evaluated LLMs. However, since this preliminary work focuses on evaluating the effectiveness of LLMs in conditions accessible to unskilled users, the one-shot methodology is deemed the most appropriate for this study.

These findings have significant implications for the field of legal document processing automation. It is suggested that, while general purpose prompts can achieve satisfactory results across a range of document types, optimal performance may necessitate a more sophisticated approach that takes into account the specific requirements of differing agreement types and the varying capabilities of different LLMs. It is recommended that future research explore the potential of adaptive prompt structures, which have the capacity to automatically adjust according to document complexity and type.

4.4. Vulnerability analysis

The results obtained from the vulnerability assessment performed by Slither are summarized in both Table 4 and Fig. 7. The former provides a high-level overview of vulnerability impacts, which are categorized into three levels: high, medium, and low, along with compilability metrics. The latter offers a more granular view of specific vulnerability checks across different impact levels for each model, as illustrated in the accompanying bar charts.

Figs. 7 and 4 reveal a correlation between the number of vulnerabilities detected and the ability of models to generate compilable smart contracts. Claude and GPT4 demonstrate identical compilability rates of 88.3 % (106/120 contracts), corresponding to their substantially higher vulnerability detection counts. In contrast, Gemini and Mistral show significantly lower compilability rates (53.3 % and 30 %, respectively). This is reflected in their more modest performances: Gemini's generated smart contracts show minimal high-impact vulnerabilities, with only one detected, while Mistral's contracts revealed 2 high-impact issues. Consequently, the smart contracts generated with these models also demonstrated fewer medium and low-impact vulnerabilities, with 90 and 97 low-impact issues, respectively. Since Claude and GPT-4-Turbo deliver more compilable smart contracts, the vulnerability detection tool used (Slither) is able to catch more vulnerabilities than those detected on compilable smart contracts generated by Gemini and Mistral. More processed smart contracts increase the probability of detecting potential security issues. Indeed, as Table 4 points out, Slither detected 8 high-impact vulnerabilities in smart contracts generated by Claude, while found 4 in those generated by GPT-4-Turbo, along with hundreds of lower-severity issues (362 and 370 low-impact vulnerabilities, respectively). The high-impact vulnerabilities detected across the models focus on six critical security issues in smart contracts:

1. **arbitrary-send-erc20**: this vulnerability occurs when there is insufficient validation of the `from` address in the `transferFrom` functions, which could allow unauthorized users to transfer tokens belonging to others. This is one of the most consistently detected high-impact vulnerabilities across the models.
2. **arbitrary-send-eth**: a vulnerability that allows unauthorized withdrawals of Ether due to unprotected functions that can send Ether to arbitrary addresses. This represents a direct financial risk as it could lead to unauthorized drainage of contract funds.
3. **unchecked-transfer**: this occurs when the return value of `transfer`/ `transferFrom` calls is not properly checked. Since some tokens do not revert on failure but return `false` instead, this could lead to silent failures and potential token theft or loss.

Table 4
Vulnerability counter identified by Slither aggregated by LLM.

LLM	High impact	Medium impact	Low impact	Compilable
Claude	8	38	362	106/120 (88.3 %)
GPT-4-Turbo	4	48	370	106/120 (88.3 %)
Gemini	1	17	90	64/120 (53.3 %)
Mistral	2	22	97	36/120 (30 %)

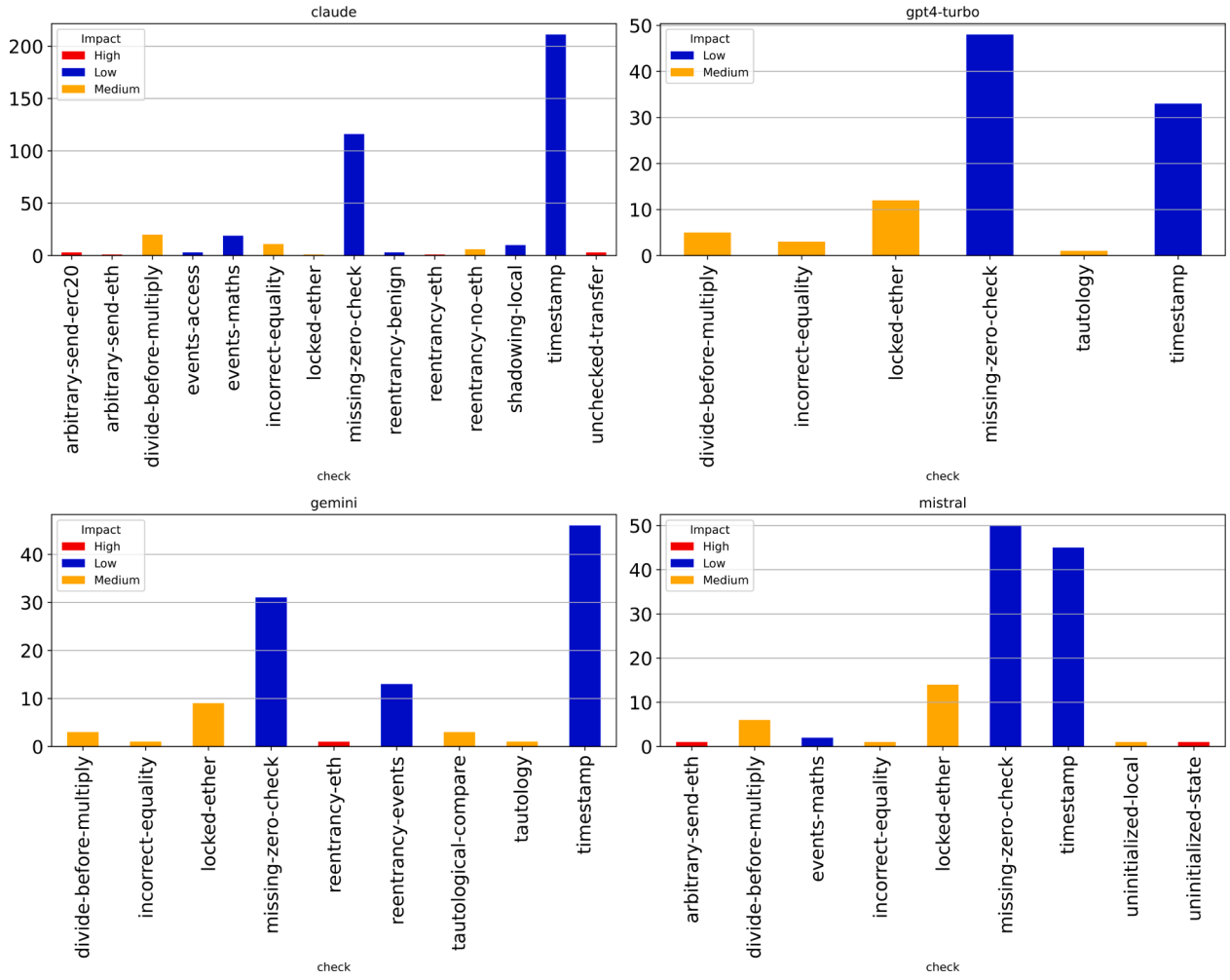

Fig. 7. Type of vulnerability detected by Slither in the generated smart contracts.

4. **encode-packed-collision**: this vulnerability occurs when `keccak256` is used to obtain a hash from `abi.encodePacked` that includes more than one dynamic type, leading to hash collisions.
5. **uninitialized-state**: this vulnerability occurs when state variables are not explicitly initialized, defaulting to zero and potentially causing critical issues. To prevent unintended behaviors, programmers should always initialize variables explicitly, even if the intended value is zero.
6. **reentrancy-eth**: this vulnerability occurs when a contract allows reentrant calls during Ether transfers, enabling attackers to manipulate the contract's state and withdraw more funds than intended. To mitigate this, programmers should use the check-effects-interactions pattern to update the state before transferring Ether.

Looking at the detection patterns, Claude's generated contracts exhibit the highest number of critical vulnerabilities, with 8 detected, followed by GPT-4-Turbo with 4 vulnerabilities. Mistral and Gemini show fewer instances, detecting 2 and 1 vulnerabilities, respectively. The higher detection rates registered on the smart contract generated by Claude and GPT-4-Turbo correlate with their superior compilability rates (88.3 %), suggesting that better code generation capabilities allow for a more detailed security analysis. It is worth mentioning some of the medium impact vulnerabilities such as `divide-before-multiply`, `incorrect-equality`, and `locked-ether` which may lead to malfunctions and misuse of the smart contract. The eager reader can refer to the detectors[14] implemented in Slither.

The prevalence of these vulnerabilities across generated smart contracts likely stems from the training data utilized in developing these Language Models. The training datasets presumably contain a significant proportion of vulnerable smart contracts, which may have influenced the models to reproduce similar security patterns. This underscores a critical consideration. While LLMs demonstrate impressive capabilities to generate smart contracts, their output inherently reflects the security flaws present in their training data. Therefore, it is imperative to perform thorough security audits and vulnerability assessments on any LLM-generated smart contracts before deployment, regardless of the model's perceived capability or reliability.

4.5. Notable patterns

An interesting pattern emerges in the results shown in Figs. 2 and 3 where some models produced technically compilable smart contracts that did not implement any of the required functionalities. This is particularly evident in several cases across different types of agreement. For instance, in the *Equipment Rental* agreement, Mistral's model achieved a 50 % compilability rate in certain prompts (PR12, PR16) while showing null values across all functional metrics - indicating that while the generated code is syntactically correct and could compile, it completely fails to implement any of the required Business Logic, Authorization, or Time Constraint patterns. Similar patterns can be observed with Gemini's responses to Property Sale agreements, where some prompts yield compilable contracts (25-50 % rate) but with zero satisfaction of the actual requirements. This phenomenon highlights an important distinction between syntactic correctness and functional completeness in smart contract generation. While achieving compilable code is a necessary first

[14] https://github.com/crytic/slither?tab=readme-ov-file#detectors

step, it is insufficient to create practically useful smart contracts. These cases underscore the importance of evaluating LLMs not just on their ability to generate syntactically correct Solidity code, but also on their capacity to properly implement the required business logic and security patterns specified in the original legal agreements.

Another notable finding emerges regarding the implementation patterns of security-related features in smart contracts generated from legal agreements. Specifically, patterns such as Fork Check Helper, Oracle integration, Randomness generation, and Pool management consistently exhibit null scores across all evaluated language models. This phenomenon can be attributed to the absence of explicit terminology or requirements in the source legal documents that would necessitate the implementation of these blockchain-specific security patterns. The result underscores a fundamental disconnect between the traditional language of the legal agreement and the specific security features of the blockchain, highlighting how conventional legal documents may not naturally encode certain crucial aspects of the implementation of secure smart contract. This observation raises important considerations for the future development of legal agreement templates specifically designed for smart contract generation, suggesting the potential need to explicitly incorporate language that maps to these essential security patterns in blockchain-based implementations.

Analysis of compilation errors across the four leading LLMs - Claude, GPT-4-Turbo, Gemini, and Mistral - in generating smart contracts reveals distinct patterns in their capabilities and limitations. Notably, Mistral exhibits the highest frequency and diversity of errors, with particular challenges in handling expression values and type conversions, accounting for 22 instances of “Expression has to be an lvalue” errors indicates a fundamental misunderstanding in how the LLM handles function parameter mutability and storage locations in Solidity. Moreover, Mistral presents 18 cases of uint256 conversion issues. Claude’s primary challenge appears to be compiler version compatibility, suggesting a potential temporal gap in its training data regarding Solidity versions. GPT-4-Turbo demonstrates relatively fewer but still significant issues, particularly with identifier declarations and deprecated syntax usage, indicating a need for updated knowledge of current Solidity best practices. Gemini shows a distinctive pattern of dependency-related errors, particularly with OpenZeppelin contracts, suggesting challenges in handling external library implementations. These findings highlight the current limitations of LLMs in producing deployment-ready smart contracts and underscore the necessity for robust post-generation validation and testing processes in practical applications.

4.6. Chain-of-thought prompting

In light of the limitations observed in the one-shot methodology, particularly with respect to functional completeness and logical precision, an additional exploratory analysis was performed using the Chain-of-Thought (CoT) prompting strategy. As previously stated, while this study primarily adopts a one-shot approach to simulate the experience of non-expert users, alternative methodologies may significantly enhance the performance of Large Language Models in smart contract generation. To substantiate this hypothesis and demonstrate the potential benefits of more structured prompting techniques, this Section presents a preliminary investigation into the effectiveness of the CoT methodology. The goal is not only to compare the results with those obtained through the one-shot approach but also to lay the groundwork for future research aimed at refining LLM-based smart contract development strategies.

In general, CoT involves a two-stage process:

- **Reasoning extraction**, in which the model generates intermediate reasoning steps to expand the context.
- **Answer extraction**, where the enriched context is used to formulate the final result or solution.

The CoT prompting strategy relies on the generation of intermediate results that are then incorporated into subsequent prompts. Typically, to improve the results, the user provides the model with examples that illustrate the desired outcome. For instance, to correct a bug in the code, the model may go through multiple phases: in the first phase, the model could receive as input an example of how such correction should be performed, in a similar context; then, the model would apply that knowledge to fix the actual bugged code. The better structured the example is, the better the outcome of the LLM becomes. CoT prompting has been successfully applied in various computer science domains (Ding et al., 2025; Duan et al., 2024; Gu et al., 2025), often yielding better results than standard one-shot or n-shot techniques. However, in the emerging field of integrating smart contracts within legal agreements, there is currently a lack of examples of legal texts alongside their corresponding smart contracts. This scarcity limits the applicability of CoT, as it typically requires illustrative examples to guide the model effectively. Despite this limitation, in addition to the one-shot analysis provided in the previous Sections, this study aims to investigate whether CoT can still improve smart contract generation by providing structured guidance. To test this hypothesis, an experiment was conducted in which the Mortgage agreement, the legal document that challenged LLMs the most when converted into a smart contract, was given to MistralAI, the LLM that performed the worst in the initial assessment, as the previous Section showed. This experiment was carried out to demonstrate that CoT reasoning can yield better results even when dealing with complex legal agreements and using LLMs with limited capabilities and using poorly formulated prompts, such as PR14 as Fig. 5 shows. The adopted methodology implements a sequential N-prompt chain that systematically guides the model through contract requirement analysis, structural planning, and security-focused refinement using the PR14 prompt as the foundation. This progressive refinement process allows the model to incrementally build understanding, rather than attempting to address all aspects simultaneously, leading to better results in terms of implemented patterns and satisfied requirements.

The CoT prompt used in this study is composed of the following components, derived by the prompt that yield the lowest performance among the considered, so PR14:

- **Role assignment and contextual framing**: The prompt begins by assigning the model the role of a senior Solidity developer, thereby priming it to adopt a professional and technically accurate tone. It also specifies that the resulting smart contract will be delivered to another senior developer, reinforcing the expectation of high-quality production-ready code.
- **Initial reasoning**: As part of the CoT approach, the model first showed an example of interaction. A rental legal agreement is provided. Subsequently, the assistant role in the prompting chain simulates how the LLM would respond to this input, generating the corresponding smart contract. This illustrates the expected behavior and structure that guide the model in handling similar tasks. Importantly, while the content appears to be system-generated, it reflects what the LLM would produce when prompted by the user.
- **Target task presentation**: The actual task is then introduced: to generate a smart contract from a mortgage legal agreement.
- **Detailed functional requirements**: To effectively guide the reasoning process, the model is given a comprehensive list of functional and technical requirements. These elements serve as intermediate reasoning steps, prompting the model to think systematically about each component before assembling the final contract.
- **Implicit multi-step reasoning**: Although not explicitly separated into multiple prompts, the detailed nature of the instructions encourages the model to simulate multi-step reasoning internally – first understanding the legal text, then mapping clauses to Solidity constructs, and finally ensuring that all technical constraints are satisfied.

- **Expected output quality**: The prompt emphasizes that the resulting contract must be fully defined and compilable, secure, ready to deploy, and structured with helper functions.

The used CoT prompt can be found below.

System: You are a senior Solidity developer. Be professional and use a formal style. You will deliver the smart contract to another senior developer.

User: Here the legal agreement: < rental agreement on Github[15] >

Assistant: Here the respective smart contract: < rental smart contract on Github[16] >

User: Here the mortgage legal agreement: < mortgage agreement on Github[17] >

User: Write a smart contract in Solidity. The software requirements are:

- target blockchain: Ethereum
- Solidity pragma >0.8
- fully defined function logic
- assign value of available parameters
- ready to deploy
- compilable
- secure

The response from MistralAI can be found on GitHub[18] and the generated smart contract is reported in Appendix A for convenience. As can be easily demonstrated, the generated smart contract is compilable. This is notable despite the low compilation rate of MistralAI in all legal agreements considered, as shown in Fig. 3. In addition, the contract exhibits a strong structure and implements a significant number of the requested features. It also can be noted that the Token pattern is implemented through the USDT integration as a transferable asset, using the ERC20 Interface to receive payments and manage funds. This aspect is important because, although the smart contract does not create its own token, the use of USDT is relevant for payment tracking purposes. The implementation of the Token pattern represents a significant improvement when using the CoT approach, as almost all smart contracts generated using the one-shot prompting technique fail to implement this pattern. Concerning the Authorization pattern, it can be seen that it is well structured with access controls based on specific roles (i.e., the *mortgagor*, the *mortgagee*, and the *owner*). The smart contract defines the *onlyMortgagor*, *onlyMortgagee*, and *validUSDTAllowance* modifiers to apply and maintain control over critical operations (i.e., receiving payments, making payments, terminating mortgages, managing emergencies). Among the modifiers, the contract defines the modifier *inActiveState* which limits operations according to the active and inactive status of the smart contract. The use of the *inActiveState* modifier supports the implementation of the Time Constraint pattern that is intended to handle payment delays and apply penalties. The *_isLatePayment* function checks whether the payment occurs beyond the *nextPaymentDue* and, if so, applies a penalty (*lateFee*). In terms of the Math pattern, the contract implicitly adopts it by exploiting the native protections of Solidity >0.8 and benefiting from automatic overflow and underflow checking. Since simple arithmetic logic is executed on the contract, there is no need for advanced mathematical libraries (SafeMath). The Termination pattern logic is implemented through the definition of the *terminateMortgage* function, which sets the *isActive* contract state to *false*, preventing further operational interactions. In addition, the contract inherits from Pausable, allowing temporary interruption of critical functions in an emergency (*pause() / unpause()*. Although the smart contract proposes a functional deactivation logic, the termination pattern results to be partially implemented since it does not contains a *kill* logic to completely deactivate the smart contract, e.g. via *selfdestruct* function. The data structures are managed according to the Helper Pattern, which is fully implemented. The contract includes the functions getMortgageDetails, *getPaymentsCount*, *getPayment*, *getTotalPaid*, *getTotalLateFees*, *getPrepaymentAmount*, *getContractBalance*, *getContractAddress*, *getUSDTTokenAddress*, which return the complete mortgage status, payment history, and funds in the smart contract, respectively, guaranteeing clarity and traceability. It makes sense to positively emphasize the implementation of the Authorization, Termination, Time Constraint and Helper patterns since, as shown in Fig. 3, Mistral's performance through the one-shot prompting technique in the mortgage contract was almost totally unsuccessful with each of the prompts utilized. Regarding the *Oracle*, *Randomness*, *Pool* or *Fork* check patterns are not present. As already said in the previous Section, this behavior can be attributed to the absence of explicit terminology or requirements in the source legal documents. In conclusion, the analysis of the smart contract and the verification of the implementation of the various patterns show that this smart contract implements more requirements than the best version of the smart contract obtained by the one-shot prompt methodology in terms of patterns and requirements for each pattern, in particular in the case of Token, Authorization, Helper and Time Constraint patterns. Finally, this test demonstrated that even a model with limited capabilities in smart contract generation, such as MistralAI, which performed poorly in implementing the patterns as shown in Fig. 6, can achieve strong and consistent results when guided through a structured CoT prompting strategy, even when dealing with complex legal agreements like the Mortgage agreement analyzed in the previous section.

4.7. Risk for automation bias and human-in-the-loop

Large Language Models are engineered to predict the most probable response from input strings. The "black-box" nature of LLMs engenders challenges in terms of response reliability (Fastowski & Kasneci, 2025). Despite receiving reinforcement learning from repetitive user feedback (Carta et al., 2023) and significant algorithmic improvements made to address AI hallucinations (Fan et al., 2023), LLMs continue to be susceptible to knowledge fabrication. Fields such as healthcare (Lee, 2024) and scientific writing (Alkaissi & McFarlane, 2023) could require a greater investment of time to detect hallucinations in the LLM response compared to code generation. In general, if the code does not function properly, the user can have a proof that the LLM hallucinates. However, if the code compiles, developers may uncritically accept syntactically correct but functionally flawed code generated by LLMs. This bias manifests itself in a particularly hazardous way in the context of blockchain, where inaccuracy or vulnerability introduced in the smart contract generated by LLMs could result in substantial financial losses (Fatima Samreen & Alalfi, 2020). This risk is exacerbated by LLMs' confident presentation of outputs and their ability to produce code that appears comprehensive. Furthermore, the more impressive an LLM's performance on simple contracts is, the greater the potential for unwarranted trust when handling complex agreements with intricate conditional logic or financial operations could be. Automation bias in the context of smart contract generation extends beyond individual errors, with the potential to undermine the broader adoption of automated contract generation tools should prominent failures occur. Consequently, effective mitigation strategies must incorporate explicit uncertainty indicators, mandatory human review protocols, and education about the particular limitations of LLM in translating legal semantics into executable code. However, with respect to this solution, it should be noted that Choi et al. highlighted that experts could still be anchored to and biased by LLM outputs (Choi et al., 2024). Hence, to further mitigate the risk of automation bias, the use of LLMs could be combined with the use of other tools, such as templates. This combination could result in two different solutions: a first solution in which the LLM is exploited to develop smaller portions of code (e.g.,

[15] https://github.com/BChain4all/pipeline/blob/main/old_test_contract_txt/LeaseAgreement.txt

[16] https://github.com/BChain4all/pipeline/blob/main/old_test_contract_txt/rental%20Agreement-smart%20contract.sol

[17] https://github.com/BChain4all/pipeline/blob/main/test_contracts_txt/mortgage_note_template.txt

[18] https://github.com/BChain4all/pipeline/blob/main/Chain-of-Thought-Results/mortgage-CoT-SC-generated.txt

that implement a small number of functions) belonging to a smart contract already presenting a verified skeleton. In this view, the skeleton could be written by experts and could import state-of-the-art security libraries, and the LLM could be used to insert additional (less relevant) functions. Another solution could let the LLM perform a classification task. In particular, starting from a set of secure templates developed by experienced programmers, the LLM could read them, by classifying them based on their topic, and then could suggest the most suitable template based on a legal contract. This solution probably could not map all the differences in a contract, but could at least result in the choice of a smart contract that does not contain vulnerabilities.

5. Conclusions

This paper presents a comprehensive evaluation of smart contract generation capabilities across leading LLMs such as Claude, GPT-4-Turbo, Gemini, and Mistral, carried out on 5 different legal contracts belonging to Cleaning Service, Compensation, Equipment Rental, Mortgage and Property Sale contexts. The analysis of the results reveals both promising advances and significant challenges in automated legal-to-code transformation. The presented analysis demonstrates a clear performance hierarchy among the evaluated models, with Claude and GPT-4-Turbo consistently outperforming Gemini and Mistral on multiple metrics. This performance gap becomes particularly pronounced when handling complex legal agreements such as mortgages and property sales.

Several key findings emerge from our investigation. First, the relationship between contract complexity and model performance appears non-linear, with even top-performing models showing marked degradation when handling intricate legal structures. Second, while achieving syntactic correctness (compilability) is increasingly feasible, ensuring functional completeness and security remains a significant challenge. This is evidenced by the concerning number of high-impact vulnerabilities detected across all generated contracts, including those from the best-performing models.

The prompt analysis reveals an interesting tension between instruction specificity and performance. While later-generation prompts (PR15-PR17) generally showed improved results, their effectiveness varied significantly across different models and agreement types. This suggests that optimal prompt design may need to be tailored both to the specific model and to the complexity of the target agreement.

The vulnerability analysis raises important concerns about the security implications of automated smart contract generation. The detection of critical vulnerabilities, even in otherwise well-performing contracts, underscores the need of robust post-generation security auditing. The correlation between higher compilability rates and increased vulnerability detection presents a paradox that warrants careful consideration in practical applications.

These findings have several important implications for both research and practice:

1. the current state of the art LLMs, while promising, are not yet suitable for unsupervised deployment in production environments, particularly for complex legal agreements;
2. the development of specialized prompt engineering techniques that account for both model capabilities and agreement complexity could significantly improve generation quality;
3. the integration of security-aware generation patterns into model training could help address the prevalence of vulnerabilities in generated contracts.

Looking ahead, several promising directions for future research emerge:

1. investigation of hybrid approaches combining template-based methods with LLM generation to enhance security and reliability;
2. development of specialized fine-tuning techniques focused on blockchain-specific security patterns and legal compliance requirements;
3. exploration of multi-step generation processes that incorporate automated validation and security checking between generations.
4. integration of the CoT prompting technique, given the promising outcomes achieved.

In conclusion, while the current generation of LLMs shows remarkable potential in bridging the gap between legal agreements and smart contracts, significant work remains to be done to address security concerns and handle complex agreement structures. The path forward likely involves a combination of improved model architectures, sophisticated prompt engineering, and robust validation frameworks. As this field continues to evolve, maintaining a balance between accessibility and security will be crucial to realize the full potential of automated smart contract generation.

CRediT authorship contribution statement

Emanuele Antonio Napoli: Investigation, Software, Formal analysis, Data curation, Methodology, Writing – original draft; **Noemi Romani:** Formal analysis, Data curation, Visualization; **Valentina Gatteschi:** Funding acquisition, Project administration, Visualization, Supervision, Writing – review & editing; **Claudio Schifanella:** Funding acquisition, Project administration, Supervision, Writing – review & editing.

Data availability

Data used for analysis can be found at https://github.com/BChain4all/pipeline.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Valentina Gatteschi reports financial support was provided by Italian Ministry of Education, University and Research. Claudio Schifanella reports financial support was provided by Italian Ministry of Education, University and Research. Noemi Romani reports financial support was provided by Italian Ministry of Education, University and Research. Valentina Gatteschi reports a relationship with Italian Ministry of Education, University and Research that includes: funding grants. Claudio Schifanella reports a relationship with Italian Ministry of Education, University and Research that includes: funding grants. Noemi Romani reports a relationship with Italian Ministry of Education, University and Research that includes: funding grants. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

The work discussed in this paper has been supported by the B4A - Blockchain for All project (https://www.blockchain4all.it/), ref. no. 20225MN5K3. This project has been funded with support from the Ministry of Education, University and Research. This document reflects the views only of the authors, and the Ministry of Education, University and Research cannot be held responsible for any use which may be made of the information contained therein.

Supplementary material

Supplementary material associated with this article can be found in the online version at 10.1016/j.eswa.2025.129011

Appendix A. Used prompts

PR12

You are a senior Solidity developer. Write a smart contract in Solidity based on the following legal agreement <legal agreement >

Be creative and use your own style. You will deliver the smart contract to another senior developer.

PR13

You are a senior Solidity developer. Write a smart contract in Solidity based on the following legal agreement <legal agreement >

Be professional and use a formal style. You will deliver the smart contract to another senior developer.

PR14

You are a senior Solidity developer. Write a smart contract in Solidity based on the following legal agreement <legal agreement>

The software requirements are:

- target blockchain: Ethereum
- Solidity pragma >0.8
- fully defined function logic
- assign value of available parameters
- ready to deploy
- compilable
- secure.

Be professional and use a formal style. You will deliver the smart contract to another senior developer.

PR15

You are a senior Solidity developer. Write a smart contract in Solidity that reflects payments, conditions, constraint, and termination of the following legal agreement <legal agreement>

The software requirements are:

- target blockchain: Ethereum
- Solidity pragma >0.8
- fully defined function logic
- assign value of available parameters
- ready to deploy
- compilable
- secure.

Be professional and use a formal style. You will deliver the smart contract to another senior developer.

PR16

You are a senior Solidity developer. A company will pay you $500,000 for the completion of the project. Write a smart contract in Solidity that reflects payments, conditions, constraint, and termination of the following legal agreement

<legal agreement>

The software requirements are:

- target blockchain: Ethereum
- Solidity pragma >0.8
- fully defined function logic
- assign value of available parameters
- ready to deploy
- compilable
- secure.

Be professional and use a formal style. You will deliver the smart contract to another senior developer.

PR17

You are a senior Solidity developer. A company will pay you $500,000 for the completion of the project. Write a smart contract in Solidity that reflects payments, conditions, constraint, and termination of the following legal agreement

<legal agreement>

The software requirements are:

- target blockchain: Ethereum
- Solidity pragma >0.8
- fully defined function logic
- assign value of available parameters
- ready to deploy
- compilable
- secure.

Be professional and use a formal style. You will deliver the smart contract to your company's legal department.

Appendix B. Smart contract pattern requirements

Table B.5
Pattern requirements definitions for Property Sale and Mortgage Note Agreements.

Patterns	Property SaleAgreement	Mortgage Note Agreement
Token	The smart contract implements payment though stable coins	The smart contract implements payment though stable coins
Authorization	The smart contract requires that the main functions can be called only by the right party.	The smart contract requires that the main functions can be called only by the right party.
Oracle	The smart contract implements calls to some oracle	The smart contract implements calls to some oracle
Randomness	The smart contract uses some random function	The smart contract uses some random function
Pool	The smart contract implement a pool mechanism	The smart contract implement a pool mechanism
Math	Mathematical operations are performed safely, with overflow check.	Mathematical operations are performed safely, with overflow check.
Business Logic	The smart contract should model: 1. Down Payment 2. Earnest Money Payment 3. Earnest Money Return 4. Buyer providing financing approval 5. Seller providing insurance policy and legal description 6. Conducting Inspections 7. Contract's termination 8. Start dispute and dispute resolution	The smart contract should model: 1. Monthly Installment's date setting 2. Monthly Installment's payment 3. Contract's termination 4. Presence of a guarantor 5. Start dispute and dispute resolution
Fork Check Helper	Check for a blockchain fork.	Check for a blockchain fork.
Helper	The smart contract should implement the following helpers. STRUCTURE: 1. Buyer 2. Seller 3. Property 4. Payment's conditions (purchase price, down amount, earnest money) FUNCTION: 1. get contract informations (parties, property) 2. get payments' details 1. Mutual Acceptance of the agreement 2. Down payment made 3. Down payment returned 4. Contract Initiated 5. Contract terminated	The smart contract should implement the following helpers. STRUCTURE: 1. Mortgagor 2. Mortgagee 3. Payment's conditions 4. Guarantor 5. Collateral property/asset FUNCTION: 1. get contract information (parties, due dates) 2. get mortgage payments' details EVENTS: 1. Monthly Installment paid 2. contract termination 3. Full balance of the Mortgage Note Repaid
Time Constraint	The smart contract implements the following: 1. on "DOWN PAYMENT", the Buyer should pay it within a certain "N_DAYS" days of mutual acceptance of this Agreement. 2. on "FINANCING APPROVAL", the Buyer should provide it to the Seller within a certain "N_DAYS" days from the contract's "START_DATE". 3. on "EARNEST MONEY PAYMENT", the Buyer should pay it to the Seller within a certain "N_DAYS" of mutual acceptance of this Agreement. 4. on "EARNEST MONEY RETURN", the Seller should pay back it to the Buyer within the transaction's "CLOSING_DATE". 5. "TRANSACTION CLOSING" should be performed at a specific date. 6. on "INSPECTION CONTINGENCY", the Buyer has until a "SPECIFIC_DATE" to conduct inspection. 7. on "INSURANCE POLICY", and "PROPERTY'S LEGAL DESCRIPTION", the Seller should provide it to the Buyer within the transaction's "CLOSING_DATE".	The smart contract implements the following: 1. on "DUE DATE", the full balance of the Mortgage Note should be paid 2. "FIRST INSTALLMENT'S PAYMENT" should be performed at a specific date and time 3. "MONTHLY INSTALLMENTS" are due within the monthly due date
Termination	Termination of the smart contract is properly handle: can't be performed any call to the smart contract functions.	Termination of the smart contract is properly handle: can't be performed any call to the smart contract functions.

Table B.6
Pattern requirements definitions for Equipment Rental, Compensation, and Cleaning Service agreements.

Patterns	Equipment Rental Agreement	Compensation Agreement	Cleaning Service Agreement
Token	The smart contract implements payment though stable coins	The smart contract implements payment though stable coins	The smart contract implements payment though stable coins
Authorization	The smart contract requires that the main functions can be called only by the right party.	The smart contract requires that the main functions can be called only by the right party.	The smart contract requires that the main functions can be called only by the right party.
Oracle	The smart contract implements calls to some oracle	The smart contract implements calls to some oracle	The smart contract implements calls to some oracle
Randomness	The smart contract uses some random function	The smart contract uses some random function	The smart contract uses some random function
Math	Mathematical operations are performed safely, with overflow check.	Mathematical operations are performed safely, with overflow check.	Mathematical operations are performed safely, with overflow check.
Business Logic	The smart contract should model: 1. Invoice's generation (make sure that late fee and every cost is applied as needed) 2. Invoice's payment 3. Contract's termination 4. Contract's renewal 5. Dispute's report 6. Dispute's resolution 7. Deposit payment 8. Deposit return (with interest rate) 9. Equipment lease renewal	The smart contract should model: 1. salary payment 2. benefit payment 3. contract termination 4. ntract extension	The smart contrat should model: 1. pay invoice 2. generate invoice 3. Contract end 4. Contract extension 5. Fault report 6. Fault resolution
Fork Check Helper	Check for a blockchain fork.	Check for a blockchain fork.	Check for a blockchain fork.
Pool	The smart contract implement a pool mechanism	The smart contract implement a pool mechanism	The smart contract implement a pool mechanism
Helper	The smart contract should implement the following helpers. STRUCTURE: 1. Lessor 2. Lessee 3. Rental price (daily rental price, additional costs, security deposit, late fee per each day the equipment has not been returned) 4. Equipment FUNCTION: 1. get contract information (parties, equipment, rental price, etc) 2. get payment details 3. get equipment list EVENTS: 1. invoice generated 2. invoice paid 3. contract termination 4. equipment returned 5. security deposit paid 6. security deposit returned	The smart contract should implement the following helpers. STRUCTURE: 1. Employer FUNCTION: 1. get contract informations 2. get payslips EVENTS: 1. payslip emission 2. benefit emission 3. contract termination	The smart contract should implement the following helpers. STRUCTURE: 1. Services 2. Invoices FUNCTIONS: 1. get contract detail 2. get services offered 3. get invoices EVENTS: 1. Invoice generated 2. Invoice paid 3. Inspection required 4. Inspection resolved 5. Contract terminated
Time Constraint	The smart contract implements the following: 1. Security deposit's payment before the contract is activated 2. Returning equipment on the due date 3. Security deposit's refund after the due date 4. Invoice must be generated on a monthly basis basis	The smart contract implements the following: 1. salary payment every month	The smart contract implements the following: 1. invoice emission 2. invoice payment 3. contract termination
Termination	Termination of the smart contract is properly handle: can't be performed any call to the smart contract functions.	Termination of the smart contract is properly handle: can't be performed any call to the smart contract functions.	Termination of the smart contract is properly handle: can't be performed any call to the smart contract functions.

References

Alkaissi, H., & McFarlane, S. I. (2023). Artificial hallucinations in ChatGPT: Implications in scientific writing. *Cureus*, *15*(2), e35179. https://www.cureus.com/articles/138667-artificial-hallucinations-in-chatgpt-implications-in-scientific-writing#!/.

Balaji, S., Dantu, R., Upadhyay, K., & McCullough, T. (2024). Bridging the legal divide: Contractual enforceability and acceptability in the AI-driven automated conversion of smart legal contracts. In *2024 IEEE 6th International conference on trust, privacy and security in intelligent systems, and applications (TPS-ISA)* (pp. 128–137). https://doi.org/10.1109/TPS-ISA62245.2024.00024

Baralla, G., Ibba, G., & Tonelli, R. (2024). Assessing github copilot in solidity development: Capabilities, testing, and bug fixing. *IEEE Access*. https://iris.polito.it/handle/11583/2990918?mode=complete.

Barbàra, F., Napoli, E. A., Gatteschi, V., & Schifanella, C. (2024). Automatic smart contract generation through LLMs: When the stochastic parrot fails. In Bartoletti, M., Schifanella, C., & Vitaletti, A. (Eds.), *Proceedings of the Sixth Distributed Ledger Technology Workshop (DLT 2024), Turin, Italy, May 14-15, 2024* (CEUR Workshop Proceedings, Vol. *3791*). CEUR-WS.org. https://ceur-ws.org/Vol-3791/paper5.pdf.

Bartoletti, M., & Pompianu, L. (2017). An empirical analysis of smart contracts: Platforms, applications, and design patterns. In M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. A. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, & M. Jakobsson (Eds.), *Financial cryptography and data security* (pp. 494–509). Cham: Springer International Publishing.

Carta, T., Romac, C., Wolf, T., Lamprier, S., Sigaud, O., & Oudeyer, P.-Y. (2023). Grounding large language models in interactive environments with online reinforcement learning. In *International conference on machine learning* (pp. 3676–3713). PMLR.

Choi, A. S., Akter, S. S., Singh, J. P., & Anastasopoulos, A. (2024). The LLM effect: Are humans truly using LLMs, or are they being influenced by them instead? arXiv preprint arXiv:2410.04699

Coignion, T., Quinton, C., & Rouvoy, R. (2024). A performance study of LLM-generated code on leetcode. In *Proceedings of the 28th international conference on evaluation and assessment in software engineering* EASE '24 (p. 79–89). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3661167.3661221

Ding, H., Liu, Y., Piao, X., Song, H., & Ji, Z. (2025). SmartGuard: An LLM-enhanced framework for smart contract vulnerability detection. *Expert Systems with Applications*, *269*, 126479. https://doi.org/10.1016/j.eswa.2025.126479

Dixit, A., Deval, V., Dwivedi, V., Norta, A., & Draheim, D. (2022). Towards user-centered and legally relevant smart-contract development: A systematic literature review. *Journal of Industrial Information Integration*, *26*, 100314. https://doi.org/10.1016/j.jii.2021.100314

Duan, X., Tan, D., Fang, L., Zhou, Y., He, C., Chen, Z., Wu, L., Chen, G., Gong, Z., Luo, W., & Guan, Q. (2024). Reason-and-execute prompting: Enhancing multi-modal large language models for solving geometry questions. In *Proceedings of the 32nd ACM international conference on multimedia* MM '24 (p. 6959–6968). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3664647.3681484

Espinha Gasiba, T., Iosif, A.-C., Kessba, I., Amburi, S., Lechner, U., & Pinto-Albuquerque, M. (2024). May the source be with you: On ChatGPT, cybersecurity, and secure coding. *Information*, *15*(9). https://doi.org/10.3390/info15090572

Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., & Zhang, J. M. (2023). Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International conference on software engineering: Future of software engineering (ICSE-foSE)* (pp. 31–53). https://doi.org/10.1109/ICSE-FoSE59343.2023.00008

Fang, P., Zou, Z., Xiao, X., & Liu, Z. (2023). iSyn: Semi-automated smart contract synthesis from legal financial agreements. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis* ISSTA 2023 (p. 727–739). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3597926.3598091

Fastowski, A., & Kasneci, G. (2025). Understanding knowledge drift in LLMs through misinformation. In M. Piangerelli, B. Prenkaj, Y. Rotalinti, A. Joshi, & G. Stilo (Eds.), *Discovering drift phenomena in evolving landscapes* (pp. 74–85). Cham: Springer Nature Switzerland.

Fatima Samreen, N., & Alalfi, M. H. (2020). Reentrancy vulnerability identification in ethereum smart contracts. In *2020 IEEE International workshop on blockchain oriented software engineering (IWBOSE)* (pp. 22–29). https://doi.org/10.1109/IWBOSE50093.2020.9050260

Feist, J., Grieco, G., & Groce, A. (2019). Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International workshop on emerging trends in software engineering for blockchain (WETSEB)* (pp. 8–15). IEEE.

Franz, F., Fertig, T., Schütz, A. E., & Vu, H. (2019). Towards human-readable smart contracts. In *2019 IEEE International conference on blockchain and cryptocurrency (ICBC)* (pp. 38–42). https://doi.org/10.1109/BLOC.2019.8751309

Greenberg, K., Kitaevich, E. J., Chaudhari, S., & Kirkland, A. (2024). Analyzing contracts: State of the field, mixed-methods guiding steps, and an illustrative example. *Law & Social Inquiry*, *49*(1), 423–450. https://doi.org/10.1017/lsi.2022.82

Grigg, I. (2004). The Ricardian contract. In *Proceedings first IEEE international workshop on electronic contracting, 2004*. (pp. 25–31). https://doi.org/10.1109/WEC.2004.1319505

Gu, X., Chen, M., Lin, Y., Hu, Y., Zhang, H., Wan, C., Wei, Z., Xu, Y., & Wang, J. (2025). On the effectiveness of large language models in domain-specific code generation. *ACM Transactions on Software Engineering and Methodology*, *34*(3). https://doi.org/10.1145/3697012

Hamdaqa, M., Met, L. A. P., & Qasse, I. (2022). iContractML 2.0: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms. *Information and Software Technology*, *144*, 106762. https://doi.org/10.1016/j.infsof.2021.106762

Hao, Z., Zhang, B., Mao, D., Yen, J., Zhao, Z., Zuo, M., Li, H., & Xu, C.-Z. (2023). A novel method using LSTM-RNN to generate smart contracts code templates for improved usability. *Multimedia Tools and Applications*, *82*(27), 41669–41699. https://doi.org/10.1007/s11042-023-14592-x

Healey, E., Tan, A. L. M., Flint, K. L., Ruiz, J. L., & Kohane, I. (2025). A case study on using a large language model to analyze continuous glucose monitoring data. *Scientific Reports*, *15*(1), 1143. https://doi.org/10.1038/s41598-024-84003-0

Husein, R. A., Aburajouh, H., & Catal, C. (2025). Large language models for code completion: A systematic literature review. *Computer Standards & Interfaces*, *92*, 103917. https://doi.org/10.1016/j.csi.2024.103917

Jurgelaitis, M., Čeponienė, L., Butkus, K., Butkienė, R., & Drungilas, V. (2023). MDA-Based approach for blockchain smart contract development. *Applied Sciences*, *13*(1). https://doi.org/10.3390/app13010487

Kfir, S., & Fournier, C. (2019). DAML: The contract language of distributed ledgers. *Communications of the ACM*, *62*(9), 48–54. https://doi.org/10.1145/3343046

Khan, F., David, I., Varro, D., & McIntosh, S. (2023). Code cloning in smart contracts on the ethereum platform: An extended replication study. *IEEE Transactions on Software Engineering*, *49*(4), 2006–2019. https://doi.org/10.1109/TSE.2022.3207428

Köpke, J., Meroni, G., & Salnitri, M. (2023). Designing secure business processes for blockchains with SecBPMN2BC. *Future Generation Computer Systems*, *141*, 382–398. https://doi.org/10.1016/j.future.2022.11.013

Lee, H. (2024). The rise of ChatGPT: Exploring its potential in medical education. *Anatomical Sciences Education*, *17*(5), 926–931.

Li, P., Li, S., Ding, M., Yu, J., Zhang, H., Zhou, X., & Li, J. (2022). A vulnerability detection framework for hyperledger fabric smart contracts based on dynamic and static analysis. In *Proceedings of the 26th international conference on evaluation and assessment in software engineering* (pp. 366–374).

Li, Y., Wang, S., Ding, H., & Chen, H. (2023). Large language models in finance: A survey. (p. 374–382). Cited by: 0; All Open Access, Green Open Access, Hybrid Gold Open Access https://doi.org/10.1145/3604237.3626869

Liu, Z., He, H., Zhang, L., & Peng, C. (2024). Leveraging XAI in prompt-based ChatGPT for financial decision support. In *Proceedings of the international conference on digital economy, blockchain and artificial intelligence* DEBAI '24 (p. 255–259). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3700058.3700099

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., & Hobor, A. (2016). Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security* CCS '16 (p. 254–269). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2976749.2978309

Mao, D., Wang, F., Wang, Y., & Hao, Z., et al. (2019). Visual and user-defined smart contract designing system based on automatic coding. *IEEE Access*, *7*, 73131–73143. Conference Name: IEEE Access https://doi.org/10.1109/ACCESS.2019.2920776

Napoli, E. A., Barbàra, F., Gatteschi, V., & Schifanella, C. (2024). Leveraging large language models for automatic smart contract generation. In *2024 IEEE 48th Annual computers, software, and applications conference (COMPSAC)* (pp. 701–710). IEEE.

Petrović, N., & Al-Azzoni, I. (2023). Model-driven smart contract generation leveraging ChatGPT. In H. Selvaraj, G. Chmaj, & D. Zydek (Eds.), *Advances in systems engineering* (pp. 387–396). Cham: Springer Nature Switzerland.

Qasse, I. A., Mishra, S., Jónsson, B. T., Khomh, F., & Hamdaqa, M. (2023). Chat2Code: A chatbot for model specification and code generation, the case of smart contracts. In *2023 IEEE International conference on software services engineering (SSE)* (pp. 50–60). https://doi.org/10.1109/SSE60056.2023.00018

Renze, M. (2024). The effect of sampling temperature on problem solving in large language models. In Y. Al-Onaizan, M. Bansal, & Y.-N. Chen (Eds.), *Findings of the association for computational linguistics: EMNLP 2024* (pp. 7346–7356). Miami, Florida, USA: Association for Computational Linguistics. https://doi.org/10.18653/v1/2024.findings-emnlp.432

Scheruhn, H.-J., von Rosing, M., & Fallon, R. L. (2015). Information modeling and process modeling. In M. von Rosing, A.-W. Scheer, & H. von Scheel (Eds.), *The complete business process handbook* (pp. 515–554). Boston: Morgan Kaufmann. https://doi.org/10.1016/B978-0-12-799959-3.00025-2

Shen, X., Li, W., Xu, H., Wang, X., & Wang, Z. (2023). A reuse-oriented visual smart contract code generator for efficient development of complex multi-party interaction scenarios. *Applied Sciences*, *13*(14). https://doi.org/10.3390/app13148094

Sobania, D., Briesch, M., Hanna, C., & Petke, J. (2023). An analysis of the automatic bug fixing performance of ChatGPT. CoRR, abs/2301.08653. https://doi.org/10.48550/arXiv.2301.08653

Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., & Alexandrov, Y. (2018). SmartCheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain* (pp. 9–16).

Trager, M. H., Gordon, E. R., Breneman, A., Kim, E., & Samie, F. H. (2025). Accuracy of ChatGPT in diagnosis and management of dermoscopic images. *Archives of Dermatological Research*, *317*(1), 184. https://doi.org/10.1007/s00403-024-03729-z

Trestioreanu, L. A., Shbair, W. M., de Cristo, F. S., & State, R. (2023). Blockly2Hooks: Smart contracts for everyone with the XRP ledger and Google Blockly. In *2023 IEEE International conference on decentralized applications and infrastructures (DAPPS)* (pp. 145–150). ISSN: 2835–3498 https://doi.org/10.1109/DAPPS57946.2023.00027

Tsai, W.-T., Ge, N., Jiang, J., Feng, K., & He, J. (2019). Invited paper: Beagle: A new framework for smart contracts taking account of law. In *2019 IEEE International conference on service-oriented system engineering (SOSE)* (pp. 134–13411). ISSN: 2642–6587. https://doi.org/10.1109/SOSE.2019.00028

Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., & Vechev, M. (2018). Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security* (pp. 67–82).

Tsiounis, K., & Kontogiannis, K. (2023). Goal driven code generation for smart contract assemblies. In *Proceedings of the 2023 12th international conference on software and computer applications* ICSCA '23 (pp. 112–121). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3587828.3587846

Upadhyay, K., Dantu, R., He, Y., Salau, A., & Badruddoja, S. (2021). Paradigm shift from paper contracts to smart contracts. In *2021 third IEEE International conference on trust, privacy and security in intelligent systems and applications (TPS-ISA)* (pp. 261–268). https://doi.org/10.1109/TPSISA52974.2021.00029

Xu, Y., Qu, W., Li, Z., Min, G., Li, K., & Liu, Z. (2014). Efficient k-means++ approximation with mapreduce. *IEEE Transactions on Parallel and Distributed Systems*, *25*(12), 3135–3144.

Zhao, X., Wei, Q., Zhu, X.-Y., & Zhang, W. (2023). A smart contract development framework for maritime transportation systems. In *2023 IEEE 23rd International conference on software quality, reliability, and security companion (QRS-C)* (pp. 310–319). ISSN: 2693–9371 https://doi.org/10.1109/QRS-C60940.2023.00091

Zubair, F., Al-Hitmi, M., & Catal, C. (2025). The use of large language models for program repair. *Computer Standards & Interfaces*, *93*, 103951. https://doi.org/10.1016/j.csi.2024.103951