

Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Computer and Control Engineering (38^{th}cycle)

Analysis, search, development, reuse, and visualization of smart contracts

By

Emanuele Antonio Napoli

******

Supervisor(s):
Prof. Valentina Gatteschi, Supervisor
Prof. Fabrizio Lamberti, Co-Supervisor

Doctoral Examination Committee:
Prof. A.B. , Referee, University of...
Prof. C.D, Referee, University of...
Prof. E.F, University of...
Prof. G.H, University of...
Prof. I.J, University of...

Politecnico di Torino
2026

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Emanuele Antonio Napoli
2026

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

I would like to dedicate this thesis to my loving family

Acknowledgements

And I would like to acknowledge ...

Abstract

Smart contracts are self-executing programs deployed on blockchain platforms that automate transactions and enforce digital agreements without intermediaries. Despite their transformative potential for decentralized finance, legal automation, and supply chain governance, their adoption is constrained by three interrelated barriers: the programming expertise required to write correct Solidity code excludes non-technical users; the immutable nature of deployed contracts makes undetected vulnerabilities permanently exploitable; and the absence of accessible comprehension tools prevents stakeholders from auditing the code that governs their assets. This thesis addresses these limitations through six contributions organized around three complementary research questions, concerning vulnerability correction, automated contract generation, and visual comprehension.

First, this work investigates the application of Large Language Models (LLMs) for automatic smart contract generation from legal agreements and textual descriptions. Through comprehensive evaluation of multiple state-of-the-art LLMs including GPT-4 by OpenAI, Claude 3.5 by Anthropic, Mistral by MistralAI, and Gemini by Google, employing the CO-STAR prompt engineering methodology, the research reveals fundamental limitations in current LLM capabilities [1]. Analysis of 57 generated smart contracts demonstrates that while LLMs achieve promising compilation rates, they consistently produce production-unsafe code containing undetected logical flaws, incorrect implementation of legal requirements, and critical security vulnerabilities. The findings establish that contemporary LLMs cannot autonomously generate reliable smart contracts, highlighting the necessity for hybrid approaches combining AI assistance with expert validation.

Towards the second research question, three studies of increasing scope investigate LLM-based smart contract generation from legal agreements. A GPT-4-build generation pipeline and the CO-STAR prompt engineering methodology achieve

98.1% compilability across 17 prompt templates applied to a property rental agreement. A subsequent expert manual evaluation of 68 generated contracts reveals five categories of critical failure invisible to automated tools, including permanently frozen funds, absent deposit logic, and universal currency mismanagement. A multi-model comparative study then extends the analysis to four frontier LLMs (GPT-4-Turbo, Claude 3.5, Mistral, and Gemini) across five legal agreements of varying complexity, totaling 480 API calls. Results establish a performance hierarchy (Claude $>$ GPT-4-Turbo $>$ Gemini $>$ Mistral), demonstrate non-linear performance degradation with agreement complexity, and show that supplying a worked exemplar together with reasoning-mode inference yields substantial improvements on the weakest model–agreement combination. The collective conclusion is that current LLMs cannot autonomously generate production-ready smart contracts and that expert review is an indispensable complement to automated metrics.

Third, to address the accessibility challenge, this thesis presents an innovative visualization and analysis tool that enhances smart contract comprehension through UML-based representations and AI-powered annotations. The system integrates automated security auditing via Slither, version control mechanisms, and intelligent code analysis using LLMs. Comparative evaluation against Remix IDE using the NASA-TLX questionnaire demonstrates a statistically significant workload reduction (52.66 versus 71.74; Wilcoxon signed-rank test, $n = 36$, $p < 0.001$), with particular improvements in mental demand, temporal demand, and user frustration. The tool successfully bridges the knowledge gap for intermediate users while maintaining utility for experts, making smart contract development and analysis accessible to everyone.

Collectively, these contributions demonstrate that LLMs and visual tools can meaningfully reduce barriers to smart contract development while establishing that human oversight remains essential for production-quality outcomes.

Contents

List of Figures **xiv**

List of Tables **xvi**

1 Introduction **1**

1.1 Background and Motivation 1

1.2 The Smart Contract Development Landscape 3

1.2.1 Smart Contract Programming and Vulnerability 3

1.2.2 The Rise of Large Language Models in Software Engineering 4

1.2.3 Smart Contract Visualization 5

1.3 Research Objectives and Contributions 6

1.4 Research Methodology 9

1.5 Thesis Structure 10

2 Background **13**

2.1 Blockchain Technology 13

2.1.1 Origin and Fundamental Architecture 13

2.1.2 Consensus Mechanism 14

2.1.3 Types of Blockchain Networks 15

2.1.4 Applications and Social Impact 15

2.2 Smart Contract 16

2.2.1 Definition and Historical Context 16
2.2.2 The Ethereum Platform and Solidity 17
2.2.3 Smart Contract Lifecycle 18
2.2.4 Vulnerabilities and Security Challenges 19
2.2.5 Development Paradigms and Accessibility Barriers 20
2.3 Large Language Models 21
2.3.1 Foundation of Language Modelling and Large Language Models 21
2.3.2 Prompt Engineering 23
2.3.3 LLMs in Software Enginering 23
2.4 Convergence: LLMs, Smart Contract, and the Research Agenda . . 24

3 Evaluating ChatGPT for Smart Contract Vulnerability Correction 26
3.1 Introduction 26
3.2 Background 27
3.2.1 Smart Contracts and the Solidity Language 27
3.2.2 Vulnerability Detection Tools and Selection of Slither 28
3.3 Related Work 29
3.4 Proposed Approach 30
3.4.1 Dataset 30
3.4.2 Pipeline Design 30
3.4.3 Best-Round Selection Criterion 31
3.5 Results 32
3.5.1 Overall Correction Rates 32
3.5.2 Results by Vulnerability Severity 33
3.6 Discussion 34
3.6.1 Compilability and Output Reliability 34

3.6.2 Effect of Providing Structured Bug Descriptions 34
3.6.3 Limitations 35
3.6.4 Implications for Automated Vulnerability Management . . . 35
3.7 Conclusions 36

4 Leveraging Large Language Models for Automatic Smart Contract Generation 37

4.1 Introduction 37
4.2 Related Work 39
4.3 System Architecture and Component Selection 41
4.3.1 LLM Selection 41
4.3.2 Security Analysis: Selection of Slither 41
4.3.3 Target Blockchain and Language 42
4.3.4 Evaluation Metrics 42
4.4 Experimental Design 43
4.4.1 Test Document 44
4.4.2 Prompt Formulation 44
4.4.3 Temperature Analysis 48
4.4.4 Consistency Evaluation 48
4.5 Results and Discussion 49
4.5.1 Compilability Rate 49
4.5.2 Effect of the Temperature Parameter 49
4.5.3 Prompt Quality and Consistency 50
4.5.4 Discussion 51
4.6 Conclusions 52

5 Manual Expert Evaluation of LLM-Generated Smart Contracts 54

5.1 Introduction 54

5.2 Related Work 55
5.3 Experimental Methodology 57
5.3.1 Test Document and Contractual Scope 57
5.3.2 Prompt Suite and Generation Pipeline 57
5.3.3 Evaluation Criteria 58
5.4 Automated Evaluation Results 58
5.5 Expert Evaluation: Functional Analysis 59
5.5.1 Rent Payment Logic 59
5.5.2 Security Deposit Management 60
5.5.3 Late Fee Assessment 60
5.5.4 Lease Lifecycle Management 61
5.6 Expert Evaluation: Parameter Initialization 62
5.7 Expert Evaluation: Code Comments 62
5.8 Discussion 63
5.8.1 The Failure of Automated Metrics as Proxies for Correctness 63
5.8.2 Absence of Correlation Between Prompt Specificity and Output Quality 64
5.8.3 Non-Detectable Logic Errors and Their Implications 65
5.8.4 Currency Mismanagement 65
5.9 Conclusions 66

6 Light and Shadows of Smart Contract Development with LLMs 67
6.1 Introduction 67
6.2 Related Work 68
6.3 Methodology 69
6.3.1 Generation Pipeline and Models 69
6.3.2 Legal Agreements 70

6.3.3 Evaluation Framework 71
6.4 Results 73
6.4.1 Overall Performance and Model Ranking 73
6.4.2 Agreement Complexity and Non-linear Performance Degradation 76
6.4.3 Prompt Design and Its Effect on Generation Quality 77
6.4.4 Vulnerability Analysis 79
6.4.5 Chain-of-Thought Prompting: A Preliminary Investigation . 81
6.5 Discussion 84
6.5.1 Structural Limits of the LLM-Based Approach 84
6.5.2 Automation Bias and the Necessity of Human Oversight . . 84
6.6 Conclusions 85

7 Smart Contract Visualization: Solutions and Challenges 87
7.1 Introduction 87
7.2 Background 89
7.2.1 Software Visualization 89
7.2.2 Smart Contract Programming Languages and Visualization Targets 89
7.3 Research Methodology 90
7.4 Smart Contract Visualization Approaches 91
7.4.1 Block-Based Visual Programming 91
7.4.2 Model-Driven visualization with BPMN 93
7.4.3 Custom and Domain-Specific visualization Approaches . . . 95
7.5 Analysis and Findings 96
7.5.1 Distribution of Visual Approaches 96
7.5.2 Temporal Evolution 96

7.5.3 Visualization Category Analysis 98
7.5.4 Accessibility Requirements and UX Evaluation 99
7.5.5 Use Case Diversity 100
7.6 Discussion 100
7.7 Conclusions 101

8 Making Smart Contracts Easier to Understand 103
8.1 Introduction 103
8.2 Related Work 104
8.3 System Architecture 106
8.3.1 Codec Microservice 106
8.3.2 Auditor Microservice 107
8.3.3 Database Microservice 108
8.3.4 AI Assistant Microservice 108
8.4 User Interface and Visualization 108
8.5 Experimental Evaluation 111
8.5.1 Study Design 111
8.5.2 Participants and Protocol 112
8.6 Results 113
8.6.1 Task Completion Time 113
8.6.2 NASA-TLX Workload Results 113
8.6.3 Discussion 116
8.7 Conclusions 117

9 Conclusions 119
9.1 Overview 119
9.2 Summary of Contributions 120

9.2.1 LLM-Based Vulnerability Correction 120
9.2.2 Benchmarking LLM-Based Smart Contract Generation . . . 121
9.2.3 The Limits of Automated Evaluation: Expert Manual Assessment 121
9.2.4 Comparative Multi-Model Evaluation 122
9.2.5 Systematic Mapping of Smart Contract Visualization Research 123
9.2.6 A Smart Contract Comprehension Tool with User Evaluation 124
9.3 Cross-Cutting Conclusions 125
9.4 Limitations 126
9.5 Future Research Directions 127
9.6 Closing Remarks 129

References **131**

Appendix A Used Prompts **148**

List of Figures

4.1 Temperature effect on Weighted Total Score computed on the automatically generated smart contracts. 50

4.2 Weighted Total Score boxplot about *4* calls to ChatGPT's API with the same prompt. 51

6.1 Pipeline used to generate smart contracts from legal agreements. . . 70

6.2 Aggregated results obtained by averaging the score of each pattern of smart contracts generated by Claude and Gemini. 74

6.3 Aggregated results obtained by averaging the score of each pattern of smart contracts generated by GPT-4-Turbo and Mistral. 75

6.4 Aggregated pattern results for each LLM used 78

6.5 Variation of pattern score for each prompt 79

6.6 Type of vulnerability detected by Slither in the generated smart contracts. 80

7.1 Comprehensive visualization of research trends: (a) illustrates the distribution of visual approaches with BPMN (35%) being most prevalent; (b) displays visualization categories with behavior (52.5%) as the dominant focus; (c) shows accessibility requirements distribution across basic (40%), intermediate (32.5%), and advanced (27.5%) categories; (d) presents the cumulative growth of papers from 2018 to 2025; (e) details the yearly breakdown of papers by visual approach type; and (f) presents the distribution of visualization categories over time, showing an increasing focus on behavior visualization in recent years. 97

8.1 System architecture of the smart contract comprehension tool. The four microservices (Codec, Auditor, Database, AI Assistant) are coordinated by a Backend-for-Frontend layer. 107

8.2 The welcome page showing the smart contract analysis overview, with vulnerability findings grouped by severity and a tab-based navigation of contract components. 109

8.3 The contract page displaying source code alongside AI-generated natural-language descriptions of functions, enabling users to correlate code segments with their real-world purpose. 110

8.4 The interactive UML-based grid view of smart contract components, with connection lines representing functional and parental relationships and a side panel providing component-level details and inline editing. 110

8.5 NASA-TLX results for the proposed tool (blue) and Remix IDE (orange), computed from the 36 matched responses. (a) Raw ratings per dimension, with 95% confidence intervals of the mean and the significance of the per-dimension Wilcoxon signed-rank tests after Holm correction ($^{***}p < 0.001$, $^{**}p < 0.01$, n.s. not significant); (b) dimension weights derived from the 15 pairwise comparisons; (c) adjusted ratings, the product of each raw rating and its sample-level weight, which sum to the composite; (d) the weighted overall workload. Lower scores indicate lower perceived workload. 114

List of Tables

3.1 Overall correction results in code-only and bug-description modes. . 32

3.2 Fixed and added vulnerabilities in code-only mode over 140 processed contracts. Percentages computed over source counts. 33

3.3 Fixed and added vulnerabilities in bug-description mode over 135 processed contracts. Percentages computed over source counts. . . . 34

4.1 Summary of smart contract Evaluation Metrics 42

4.2 Components of the prompt templates PR12–PR17 45

6.1 Word and estimated token counts for each legal agreement (assuming 1 token $\approx$ 4 characters on average). 71

6.2 Vulnerability counts identified by Slither, aggregated by LLM, with compilability rates. 79

7.1 Selected papers in chronological order, with visual approach, category, availability, use case, accessibility requirement, and UX evaluation detail. 92

8.1 Representative low-code and no-code tools for smart contract development and analysis. The *Formalism* column groups the tools more coarsely than the systematic mapping study of Chapter 7: each label generalises the finer visual-approach classification recorded in Table 7.1, which remains the reference classification for the works surveyed there. 105

8.2 NASA-TLX raw ratings per dimension and weighted overall workload score for the proposed tool and Remix IDE. Lower values indicate lower perceived workload. The p-values are those of two-sided Wilcoxon signed-rank tests on the 36 matched pairs, Holm-corrected across the six dimensions; the overall row reports the test on the participant-level weighted composite. 116

Chapter 1

Introduction

1.1 Background and Motivation

The emergence of blockchain technology over the past decade has fundamentally altered the landscape of distributed computing, digital finance, and contract enforcement. Originally conceived by Satoshi Nakamoto in 2008 as the underlying infrastructure for Bitcoin, a peer-to-peer electronic cash system [2], blockchain technology offered a radical departure from the trust models that had governed digital transactions since the dawn of the internet. Rather than relying on centralized intermediaries such as banks, notaries, or payment processors, Nakamoto's design enabled two mutually distrusting parties to transact directly, with finality guaranteed by a distributed network of nodes maintaining a cryptographically secured ledger. The implications of this architecture extended far beyond digital currency: the same properties of immutability, transparency, and decentralization that made Bitcoin attractive as a monetary system proved to be the foundations upon which an entirely new paradigm of programmable agreements could be built.

The concept of encoding contractual logic in software predates blockchain technology by more than a decade. Nick Szabo, writing in 1994, proposed the idea of *smart contracts* [3] as "a set of promises, specified in digital form, including protocols within which the parties perform on these promises" [4]. Szabo theorized vending machines as a primitive analog: once a coin is inserted and a selection made, the machine automatically executes the agreement without requiring a human intermediary to verify and enforce it. The challenge in 1994 was that no computational

medium existed that could enforce such agreements in a tamper-proof, trustless manner. Blockchain technology provided exactly this substrate. When the Ethereum platform launched in 2015, it introduced Ethereum Virtual Machine (EVM) and Solidity programming language, enabling developers to implement Turing-complete programs, called smart contracts, directly onto a public immutable ledger [5]. From that moment forward, the smart contract ceased to be a theoretical construct and became an operational reality.

Smart contracts now underpin an ecosystem of extraordinary scope. Decentralised Finance (DeFi) protocols [6, 7], non-fungible token (NFT) marketplaces [8, 9], decentralised autonomous organisations (DAOs) [10, 11], supply chain management systems [12, 13], and digital identity platforms [14, 15] all rely on smart contract logic to automate financial and administrative processes without the need for centralised governance. According to documented measurements, the total value locked (TVL) across DeFi protocols grew from under one billion USD in early 2020 to approximately 150 billion USD by April 2022 [7], while the structural adoption of blockchain-based automation has continued to deepen despite subsequent market contractions.

The history of smart contract security incidents is sobering. The 2016 attack on "The DAO", a decentralized investment fund built on Ethereum, exploited a reentrancy vulnerability in the contract code, enabling an attacker to drain approximately 3.6 million Ether, valued at the time at roughly $60 million [16]. The response of the Ethereum community, which involved a contentious hard fork to reverse the attack, itself caused lasting controversy. In subsequent years, vulnerabilities including integer overflow, arbitrary send, unchecked return values, and timestamp dependence have been exploited in dozens of high-profile incidents, causing cumulative losses estimated to be billions of dollars [17]. These are not hypothetical risks: they are recurring, financially disastrous events whose root cause is the gap between the programming sophistication required to write correct Solidity code and the level of expertise possessed by the majority of potential smart contract authors.

This gap is structural. Solidity, the dominant language for EVM-compatible smart contracts, requires familiarity with concepts that are specific to the blockchain context, such as gas optimization, storage layout, delegate calls, fallback functions, and the precise semantics of Ether transfers. All of these are in addition to the general competencies expected of experienced software engineers. Yet, the parties who

stand to benefit most from smart contract automation are frequently non-technical: legal professionals wishing to encode binding agreements, entrepreneurs building decentralized applications, small businesses seeking to automate payment workflows, and individuals participating in DeFi protocols without the means to audit the code with which they interact. The result is a fundamental tension at the core of the blockchain ecosystem: a technology with transformative potential for broad societal adoption, gated behind a technical barrier that excludes the very population from which it could most benefit.

This thesis addresses this challenge through six contributions organized around three complementary research questions. The first question investigates whether LLMs can help with the correction of known security vulnerabilities in existing smart contracts. The second question, comprising three studies of increasing scope, examines the extent to which LLMs can generate complete smart contracts from legal agreements and develops evaluation methodologies capable of assessing the quality of such generated code. The third question turns to the comprehension side of the problem, surveying the landscape of smart contract visualization research and contributing a tool that integrates visual representation, automated security analysis, and AI-assisted annotation to reduce cognitive workload. Together, these contributions constitute a coherent program of work whose unifying aim is to make the creation, auditing, and comprehension of smart contracts accessible to a broader population of practitioners.

1.2 The Smart Contract Development Landscape

1.2.1 Smart Contract Programming and Vulnerability

The development of a smart contract in Solidity follows a process broadly analogous to conventional software engineering: a developer writes source code, compiles it to EVM bytecode, and deploys it to the blockchain. Once deployed, the contract address is immutable and the code cannot be modified. This immutability, which is central to the trust model of smart contracts, simultaneously eliminates the possibility of post-deployment patching. Unlike a web application whose back-end can be updated silently to fix a security flaw, a vulnerable smart contract remains vulnerable as long as it holds funds or processes transactions. The only recourse available to

developers is upgradeability patterns, which are commonly proxy contracts that delegate execution to a replaceable logic contract. However, this upgradeability pattern introduces its own complexity and security surface [18].

Vulnerability detection has consequently become a critical research area. Tools such as Slither [19], Mythril [20], Manticore [21], Oyente [22], SmartCheck [23], and Securify [24] approach the problem through static analysis, symbolic execution, or formal verification. Slither, developed by Trail of Bits, performs static analysis of Solidity source code by converting it to an intermediate representation (SlithIR) and applying a library of detectors corresponding to known vulnerability patterns. It is capable of identifying dozens of vulnerability classes, including reentrancy, arbitrary send, unchecked low-level calls, and incorrect ERC-20 token implementations, with low false-positive rates and sub-second execution times on typical contracts [19]. Mythril employs symbolic execution on EVM bytecode, exploring execution paths to identify reachable states corresponding to known vulnerability patterns. Each tool occupies a distinct position in the trade-off space between soundness, completeness, and practical usability.

Despite the maturity of the vulnerability detection landscape, correction of identified vulnerabilities remains largely a manual process. A developer who receives a Slither report identifying an arbitrary-send vulnerability must understand the semantics of the vulnerability, reason about the specific code path in which it manifests, and redesign the logic, which is a process requiring both security expertise and intimate knowledge of the codebase. This manual correction workflow is slow, error-prone, and inaccessible to developers without a security background. The question of whether Large Language Models (LLMs) could automate or assist this correction process is one of the central questions addressed in this thesis.

1.2.2 The Rise of Large Language Models in Software Engineering

Large Language Models, trained on vast corpora of text and code, have demonstrated remarkable capabilities in code generation, completion, explanation, and refactoring tasks [25]. Models such as OpenAI's GPT series, Google's Gemini, Anthropic's Claude, and Mistral AI's Mistral have demonstrated substantial and rapidly expanding capabilities across a broad range of software engineering tasks, including code

generation, bug detection, and automated program repair [26]. Standard evaluation frameworks, such as the SWE-bench benchmark introduced by [27], have been developed specifically to measure these capabilities against real-world software engineering problems, and progress on such benchmarks has been considerable, though significant challenges remain [27]. The widespread adoption of tools such as GitHub Copilot, which itself is built on an LLM fine-tuned on public code repositories, has begun to reshape the workflows of professional software development, and its capabilities have been assessed specifically in the smart contract setting [28].

The widespread adoption of tools such as GitHub Copilot which itself is built on an LLM fine-tuned on public code repositories, has begun to reshape the workflows of professional software development, with surveys suggesting high rates of productivity improvement among early adopters [29].

The applicability of these capabilities to smart contract development is not self-evident. Solidity is a relatively low-resource language in LLM training corpora compared to Python, JavaScript, or Java: there is substantially less Solidity code available for pre-training, and the semantics of smart contract execution differ from those of conventional programs in ways that are not always well-represented in general-purpose training data. Questions of gas efficiency, Ether accounting, the precise ordering of state updates and external calls, all of which are critical to smart contract correctness, require domain-specific knowledge that LLMs may not reliably possess. Furthermore, the evaluation of LLM-generated smart contracts presents methodological challenges: automated metrics such as compilability and static analysis scores can be computed at scale, but they are insufficient to detect logical errors that do not manifest as syntactic errors or known vulnerability patterns. As this thesis demonstrates, expert manual evaluation is an indispensable complement to automated analysis and is capable of identifying critical flaws that automated tools systematically miss [30].

1.2.3 Smart Contract Visualization

A parallel thread of research has addressed the comprehension problem from a different angle: rather than generating smart contract code, visualization tools seek to make existing smart contracts legible to non-expert users. The motivation is straightforward. Visualization bridges the gap between the formal specification of a

smart contract and an intuitive representation that communicates its structure and behavior to a human observer.

Research on smart contract visualization has grown substantially since the emergence of Ethereum, driven by the recognized difficulty of reasoning about contract logic directly from source code [31]. Early surveys of blockchain-oriented visualization tools identified a broad diversity of approaches, spanning transaction graphs, structural diagrams, and behavioral representations, while noting that tools specifically targeting smart contract comprehension remained underexplored [32]. Parallel work on low-code and model-driven development platforms highlighted BPMN and UML as the dominant formalisms to represent contract workflows, due to their familiarity to software practitioners and their ability to bridge the semantic gap between business requirements and on-chain implementations [33]. A recent systematic mapping of the dedicated smart contract visualization literature, covering forty papers published between 2018 and 2025, confirms this landscape: BPMN accounts for 35% of surveyed works, followed by UML-based representations at 20% and Blockly visual programming at 17.5% [34]. The same study identifies behavioral visualizations, those representing dynamic execution of a contract, as the dominant category at 52.5% of approaches, with structural visualizations at 37.5% and evolution-tracking approaches at 10%. Importantly, the review reveals a significant gap in empirical validation: only 20% of the surveyed works include formal usability evaluation, and those that do typically rely on a small sample size of participants, limiting the generalizability of their findings [34]. This absence of rigorous user-centered evaluation remains an open challenge in the field.

1.3 Research Objectives and Contributions

The objective of this thesis is to reduce the barrier to smart contract adoption for non-expert users through a program of research that spans the full lifecycle of smart contract development: from vulnerability correction, through automated generation, to comprehension and visual analytics. The research is organized around three research questions, each addressing a distinct facet of the accessibility gap and together yielding six contributions.

1. **Automated vulnerability correction**: Can LLMs reliably identify and correct security vulnerabilities in existing smart contracts when provided with static analysis reports? This question is addressed by a single contribution (Chapter 3).

2. **Automated smart contract generation**: Can LLMs generate correct, compilable, and secure smart contracts from natural language or legal text descriptions, and what prompt engineering strategies and evaluation methodologies are most effective? This question is addressed by three contributions of increasing scope (Chapters 4, 5, and 6).

3. **Smart contract comprehension tools**: What is the state of the art in smart contract visualization, and can a UML-based visualization tool, integrated with AI assistance and automated analysis, reduce the cognitive load experienced by non-expert users seeking to understand smart contract logic? This question is addressed by two contributions (Chapters 7 and 8).

The thesis makes the following specific contributions.

Contribution 1: Empirical evaluation of LLM-based vulnerability correction. This thesis presents the first systematic empirical study of ChatGPT (`gpt-3.5-turbo`) for smart contract vulnerability correction, evaluated on a dataset of 143 Solidity smart contracts with labeled vulnerabilities drawn from public repositories [35]. The study introduces a structured pipeline in which Slither identifies vulnerabilities and ChatGPT is prompted to correct them across four iterative rounds with varying levels of contextual information. Results demonstrate that ChatGPT produces compilable output in 89.7% of cases and achieves a full vulnerability fix in 27.1% of cases, and that including a structured description of the vulnerability raises the best-round rate of correcting at least one vulnerability from 57.1% to 58.5%, an improvement of 1.4 percentage points over prompts containing only the vulnerable code. All experimental code and data are released as open-source artifacts [36].

Contribution 2: A benchmark pipeline for LLM-based smart contract generation. This thesis introduces a reusable pipeline for the automatic generation of smart contracts from legal agreements and natural language descriptions using

LLMs [37]. The pipeline incorporates the CO-STAR prompt engineering methodology [38] and evaluates generated contracts on a benchmarking suite comprising seven quantifiable metrics: compilability, vulnerability density, Solidity version compliance, number of generated contracts and functions, comment coverage, and initialized parameter rate, aggregated into a Weighted Total Score. Evaluated on GPT-4 (`gpt-4-0125-preview`) across the six selected prompt templates (54 generations), the pipeline achieves 98.1% compilable output, and CO-STAR-structured prompts specifying a senior Solidity developer with explicit software requirements (PR15–PR17) yield the most consistent and highest-scoring results.

Contribution 3: Evidence that automated evaluation is insufficient for LLM-generated smart contracts. A critical finding of this thesis is that automated evaluation metrics, including static analysis scores and compilability rates, are insufficient to assess the correctness of LLM-generated smart contracts [30]. Through expert manual review of 68 GPT-4-generated smart contracts derived from a property rental agreement, the study identifies critical logical flaws, including funds permanently locked in the contract due to the absence of a landlord withdrawal function, systematic misinterpretation of currency denominators (USD to ETH), absent security deposit collection logic, and incomplete business-logic coverage, none of which automated tools detected. This finding constitutes a methodological contribution: it establishes expert manual evaluation as a necessary, not merely optional, component of any credible quality assessment framework for LLM-generated smart contracts, and it motivates the design of richer, specification-grounded evaluation criteria pursued in Contribution 4.

Contribution 4: A comparative evaluation of state-of-the-art LLMs for smart contract generation. This thesis presents the most comprehensive comparative evaluation of LLMs for smart contract generation reported to date, encompassing four frontier models (Anthropic Claude `claude-3-5-sonnet-latest`, OpenAI GPT-4-Turbo `gpt-4-turbo`, Google Gemini `gemini-1.5-pro`, and Mistral `mistral-large-latest`), five legal agreements of varying complexity, six prompt templates, and four independent runs per configuration, totalling 480 API calls [1]. Evaluation combines automated Slither analysis with a novel manual pattern-based scoring framework derived from the smart contract design pattern taxonomy of Bartoletti and Pompianu [39] and Khan et al. [40], extended with a new Business

Logic pattern. Results establish a performance hierarchy (Claude > GPT-4-Turbo > Gemini > Mistral), reveal a non-linear relationship between agreement complexity and model performance, and demonstrate that supplying a worked exemplar together with reasoning-mode inference substantially improves output quality for an underperforming model–agreement combination.

Contribution 5: A systematic mapping study of smart contract visualization research. This thesis contributes a systematic mapping study of 40 papers on smart contract visualization published between 2018 and 2025, characterizing the field along five dimensions: visualization methodology, representation type (behavioral, structural, evolutionary), accessibility level, use case domain, and evaluation rigor [34]. The study reveals BPMN as the dominant methodology at 35% of surveyed works, identifies a critical gap in formal usability evaluation (present in only 20% of studies), and provides a structured research agenda for the field.

Contribution 6: A UML-based smart contract comprehension tool with user evaluation. This thesis presents the design, implementation, and empirical evaluation of an open-source smart contract comprehension tool that combines UML-based structural visualization, automated vulnerability analysis via Slither, AI-assisted natural-language explanation, and a version control system for contract evolution tracking [41]. The tool is implemented as a microservice architecture using Golang and Flutter, and is evaluated through a controlled user study with 36 volunteers using the NASA Task Load Index workload assessment instrument [42]. Results show a statistically significant reduction in perceived workload compared to Remix IDE (52.66 vs. 71.74 on a 0–100 scale; Wilcoxon signed-rank test, $n = 36$, $p < 0.001$), with particularly pronounced improvements in frustration (49.17 vs. 81.39) and mental demand (61.67 vs. 78.89).

1.4 Research Methodology

The research program presented in this thesis is empirical in character, combining software engineering with experimental evaluation methodologies. The design philosophy underlying the experimental work reflects two convictions: first, that claims about the capabilities of LLMs for smart contract tasks must be grounded in

systematic, reproducible experiments with clearly defined evaluation criteria; and second, that the ultimate measure of any tool designed to assist human users is the experience of those users, requiring user-centered evaluation methods alongside technical metrics.

Several methodological choices are worth considering. The use of iterative prompt refinement, particularly the CO-STAR framework [38], as a structured approach to prompt engineering reflects the view that prompt design is not a matter of trial and error but a systematic engineering discipline with principled techniques. The adoption of the NASA Task Load Index (NASA-TLX) [42] for workload assessment in the usability study reflects a commitment to established and validated instruments from the literature on human-computer interaction rather than ad-hoc satisfaction surveys. The development of a manual scoring framework derived from the Bartoletti and Pompianu pattern taxonomy [39] reflects the conviction, validated by empirical results, that domain-specific evaluation criteria are necessary to detect classes of error that generic code quality metrics miss. Last but not least, the release of all experimental pipelines and datasets as open-source software reflects a commitment to reproducibility and the cumulative advancement of the field.

1.5 Thesis Structure

The remainder of this thesis is organized as follows.

Chapter 2: Background provides the technical foundation for all subsequent contributions. It covers the architecture of blockchain systems, the semantics of Ethereum smart contracts and the Solidity programming language, the principal vulnerability classes and their detection tools, and the capabilities and limitations of current Large Language Models for code-related tasks.

Chapter 3: Evaluating ChatGPT for Smart Contract Vulnerability Correction presents Contribution 1. It introduces a pipeline coupling Slither's static analysis with ChatGPT's generative repair capabilities, evaluates the pipeline on 143 labeled vulnerable contracts drawn from the SmartBugs-Curated benchmark, and quantifies the effect of providing structured vulnerability descriptions on fix rates.

Chapter 4: Leveraging Large Language Models for Automatic Smart Contract Generation presents Contribution 2. It introduces the CO-STAR-structured

prompt engineering pipeline and the Weighted Total Score benchmarking suite, and evaluates GPT-4 across 17 prompt templates on a property rental agreement, demonstrating that structured prompts with explicit software requirements yield the most consistent and highest-scoring outputs. The chapter also identifies the ceiling imposed by purely automated evaluation and motivates the expert analysis of the following chapter.

Chapter 5: Manual Expert Evaluation of LLM-Generated Smart Contracts presents Contribution 3. It complements the automated analysis of Chapter 4 with a structured expert manual review of 68 GPT-4-generated contracts, examining functional correctness, parameter initialization fidelity, and business-logic completeness. The chapter demonstrates that critical logical flaws, including frozen funds, absent deposit logic, missing late-fee assessment, and universal currency mismanagement, are systematically invisible to static analysis tools, and establishes expert evaluation as a necessary component of any rigorous quality assessment framework for LLM-generated smart contracts.

Chapter 6: Light and Shadows of Smart Contract Development with LLMs presents Contribution 4. It reports the comprehensive comparative evaluation of four frontier LLMs across five legal agreements and six prompt templates, introduces the manual pattern-based scoring framework derived from the Bartoletti and Pompianu design pattern taxonomy, and analyzes, in a preliminary experiment on the weakest model–agreement combination, the effect of supplying a worked exemplar together with reasoning-mode inference.

Chapter 7: Smart Contract Visualization: Solutions and Challenges presents Contribution 5. It describes the systematic mapping study of 40 papers on smart contract visualization, characterizes the methodological landscape, and identifies open research directions including the gap in rigorous usability evaluation.

Chapter 8: Making Smart Contracts Easier to Understand presents Contribution 6. It describes the architecture and implementation of the UML-based smart contract comprehension tool, details the AI-assistant and Slither integrations, and reports the results of the NASA-TLX user study demonstrating a statistically significant reduction in perceived cognitive workload relative to Remix IDE.

Chapter 9: Conclusions synthesizes the six contributions of the thesis, draws cross-cutting conclusions about the state of LLM-assisted smart contract engineering and its significance for the goal of making blockchain technology accessible to

non-expert users, discusses the limitations of the work, and identifies directions for future research.

Chapter 2

Background

2.1 Blockchain Technology

2.1.1 Origin and Fundamental Architecture

The history of blockchain technology can be traced to the seminal 2008 white paper by pseudonymous Satoshi Nakamoto, titled Bitcoin: A Peer-to-Peer Electronic Cash System [2]. In this landmark document, Nakamoto outlined a purely peer-to-peer version of electronic cash capable of enabling online payments to be sent directly from one party to another without relying on a financial institution. The proposal introduced a novel solution to the longstanding double-spending problem by exploiting a peer-to-peer network in which transactions are timestamped by hashing them into an ongoing chain of hash-based proof-of-work. The white paper constitutes the foundational specification of what would become one of the most disruptive technological paradigms of the early twenty-first century.

Distributed ledger technology has emerged as a transformative force in numerous sectors, fundamentally altering the way transactions and data are managed in decentralized environments. Blockchain technology is a system that records transactions across a network of computers in a secure and immutable manner. Each block, a batch of transactions, is linked to the previous one via cryptographic hashing, forming a chronological chain of blocks. This distributed ledger eliminates the need for a central authority, as consensus among participants ensures the validity and integrity of the data. Each participant retains a copy of the entire blockchain,

enhancing transparency and drastically reducing the risk of fraud or tampering. Since no single entity controls the ledger, the risk of a single point of failure, whether due to a technical outage or malicious attack, is substantially mitigated compared to conventional centralized database architectures.

The structural composition of a blockchain comprises three foundational layers. The data layer encodes the actual transactional information within each block, which typically includes a cryptographic hash of the previous block, a timestamp, a Merkle root of all transactions, and a nonce value used in the mining process. The network layer governs peer-to-peer communication between nodes, ensuring consistent propagation of new blocks and transactions throughout the network. Finally, the consensus layer defines the protocol by which distributed nodes agree on the canonical version of the ledger [43]. The elegant interlocking of these three layers produces a system in which the integrity of any historical record is computationally guaranteed: altering a past block requires recomputing the proof-of-work for all subsequent blocks, a task that rapidly becomes infeasible as the chain grows in length.

2.1.2 Consensus Mechanism

A consensus mechanism is the algorithmic process by which all nodes in a blockchain network reach agreement on the state of the shared ledger. The choice of consensus algorithm has profound implications for the security of a blockchain system, scalability, and energy efficiency [43]. The earliest and most well-known mechanism is Proof of Work (PoW), first described in the Bitcoin white paper [2]. In PoW, miners compete to solve computationally expensive cryptographic puzzles; the first to find a valid solution broadcasts the new block and receives a reward. Although PoW is considered highly secure against Sybil attacks, since attacking the network requires control of the majority of CPU power, it has been widely criticized for its significant energy consumption. To address the inefficiencies of PoW, alternative consensus algorithms have been proposed. Proof of Stake (PoS) selects validators proportional to the amount of cryptocurrency they stake, or lock as collateral, thereby replacing computational work with economic commitment. Ethereum, the most prominent blockchain platform that supports smart contract execution, migrated from PoW to PoS in September 2022, known as The Merge. Beyond PoW and PoS, comparative analyses of consensus algorithms [43] have cataloged numerous hybrid and derivative mechanisms, including variants of Delegated Proof of Stake (DPoS),

Proof of Authority (PoA), and Byzantine Fault Tolerant (BFT), each providing different tradeoffs between decentralization, throughput, and finality. The Byzantine Generals Problem, formally articulated by Lamport, Shostak, and Pease in 1982, provides the theoretical underpinning for these fault-tolerant designs, establishing the mathematical conditions under which distributed systems can reach consensus despite the presence of malicious or faulty actors.

2.1.3 Types of Blockchain Networks

Blockchain networks are commonly categorized along two orthogonal dimensions: permissioning model and governance structure. Public blockchains, such as Bitcoin and Ethereum, are open to any participant who can read, write, and validate transactions without prior authorization. They maximize decentralization and censorship resistance but typically exhibit lower throughput and higher latency. Private blockchains restrict participation to a pre-approved set of actors, trading decentralization for significantly higher performance and greater regulatory compliance. Consortium or federated blockchains occupy a middle ground, wherein a predefined group of organizations jointly control the network, making them common in inter-organizational business applications [44]. The evolution of blockchain from purely cryptocurrency-focused platforms (Generation 1) to programmable platforms enabling general-purpose computation (Generation 2, exemplified by Ethereum) and beyond to highly scalable, interoperable networks (Generation 3) has been comprehensively surveyed in the literature [44]. This evolution was pivotal: the introduction of Turing-complete scripting environments allowed developers to deploy arbitrary executable logic directly on-chain, giving rise to the concept of smart contracts. Generation 3 platforms, including Polkadot, Cosmos, and Algorand, address scalability challenges through techniques such as sharding, layer-2 rollups, and cross-chain interoperability protocols [45], seeking to resolve what is commonly termed the blockchain trilemma: the simultaneous achievement of decentralization, security, and high throughput.

2.1.4 Applications and Social Impact

The foundational properties of blockchain, transparency, immutability, decentralization, and trustless execution, make it uniquely suited for a broad spectrum of

applications. In supply chain management, blockchain provides an auditable, tamper-proof record of a product's provenance and handling history, enabling stakeholders to verify claims about origin, quality, and authenticity [46]. In document notarization and intellectual property management, immutable timestamps and cryptographic hash registrations provide unforgeable evidence of a document's prior existence, with applications ranging from patent registration to academic credential verification [47]. In the financial sector, blockchain enables peer-to-peer payment processing and programmable financial instruments with minimal intermediary involvement, substantially reducing transaction costs and settlement times [48, 6]. In healthcare, blockchain offers a framework for secure patient data sharing with explicit, auditable consent mechanisms, enabling interoperable electronic health records while preserving patient sovereignty over sensitive medical information [49, 50]. Decentralized Applications (dApps), which combine on-chain smart contracts with off-chain user interfaces, represent the dominant paradigm for harnessing blockchain's programmability [51]. The number of deployed dApps has grown dramatically since the launch of Ethereum in 2015, spanning domains from decentralized finance (DeFi) and non-fungible tokens (NFTs) to gaming, governance, and identity management. Despite this growth, significant challenges remain, including scalability (the so-called blockchain trilemma between decentralization, security, and scalability), interoperability between heterogeneous networks, and regulatory uncertainty [31]. The DeFi ecosystem alone had accumulated tens of billions of dollars in total value locked (TVL) by 2020 [6] and approximately 150 billion USD by April 2022 [7], underscoring both the transformative potential and the attendant financial risk inherent in deploying critical financial logic as immutable on-chain code.

2.2 Smart Contract

2.2.1 Definition and Historical Context

The concept of a smart contract precedes the blockchain technology itself. The term was coined by computer scientist and legal scholar Nick Szabo in 1994 [4], who defined a smart contract as a computerized transaction protocol that executes the terms of a contract. Szabo envisioned self-enforcing contractual clauses embedded in hardware and software, noting that digital cash protocols represented an elementary

form of smart contract. However, the realization of this vision at scale required a platform that could guarantee the trustless, deterministic execution of code, a property that blockchain uniquely provides. In the context of modern blockchain platforms, smart contracts are self-executing programs stored on the blockchain that run when predetermined conditions are met [52]. They encode the terms of an agreement directly into code, which is then deployed on-chain and executed automatically by the network's nodes whenever the specified triggering conditions are satisfied. Because the logic of the contract lies on an immutable, transparent, and decentralized infrastructure, the outcome of its execution is guaranteed to all parties without the need for intermediaries such as lawyers, notaries, or banks. Smart contracts can serve as digital twins of legal agreements, leveraging their intrinsic properties of non-repudiability of data and trustless execution. From a legal-technology perspective, they could represent a radical departure from the centuries-old paradigm of contract enforcement, replacing judicial recourse with algorithmic certainty.

2.2.2 The Ethereum Platform and Solidity

Ethereum, conceptualized by Vitalik Buterin and launched in 2015 [53], is widely regarded as the foundational platform for smart contract development. Unlike Bitcoin's limited scripting language, Ethereum provides a Turing-complete execution environment known as the Ethereum Virtual Machine (EVM) [54]. The EVM is a sandboxed, stack-based virtual machine that executes bytecode compiled from high-level smart contract languages. Any participant in the network runs an identical instance of the EVM, ensuring fully deterministic and consistent execution of contract code regardless of the underlying hardware. Solidity is the dominant high-level programming language for writing smart contracts on Ethereum and EVM-compatible blockchains [54]. It is a statically typed, contract-oriented language syntactically similar to JavaScript and C++, featuring concepts such as state variables, functions, modifiers, events, and inheritance. To deploy a smart contract on the blockchain, the Solidity source code must first be compiled by the solidity compiler (solc) into EVM bytecode. The compiled bytecode is then submitted to the Ethereum network via a deployment transaction, whereupon it is permanently stored on-chain. Accompanying the bytecode is the Application Binary Interface (ABI), a JSON descriptor defining the contract's functions, parameters, and return types, which enables external applications (dApps) to interact with the deployed

contract. A distinctive and economically significant feature of Ethereum's execution model is the concept of gas. Every EVM operation carries a fixed gas cost, and users submitting transactions must attach a gas limit and gas price denominated in Ether (ETH) [53]. This mechanism serves two purposes: it compensates miners (or validators, post-Merge) for the computational resources expended, and it prevents the halting problem by ensuring that all computations terminate within a bounded number of steps. When a transaction's gas runs out before completion, execution is reverted, preventing infinite loops from freezing the network. The gas mechanism also creates important economic incentives for code efficiency: developers who deploy computationally wasteful contracts impose higher transaction costs on their users, creating a market pressure toward optimization that has few analogues in conventional software engineering.

2.2.3 Smart Contract Lifecycle

The lifecycle of a smart contract encompasses four main phases [55]. During the design phase, developers translate business requirements that may originate from legal agreements, organizational policies, or technical specifications into formal contract logic. During the development phase, the contract is implemented in Solidity (or another smart contract language), tested against unit and integration test suites, and subjected to static analysis and formal verification where applicable. In the deployment phase, the compiled bytecode is submitted to the network; upon confirmation in a mined block, the contract receives a permanent address and begins accepting transactions. During the operational phase, the contract autonomously executes its logic in response to incoming transactions, maintaining its state on-chain. Crucially, once deployed, a smart contract's code cannot be modified, a property that confers trustlessness but also introduces significant risks when bugs or vulnerabilities are present. It should be noted that certain upgradeability patterns, such as the proxy pattern and the diamond standard, have emerged as pragmatic responses to the immutability constraint [18]. These patterns separate the contract's storage from its logic by interposing a proxy contract that delegates calls to a replaceable implementation contract. While upgradeability mitigates the risk of locked-in bugs, it introduces a degree of centralization and governance complexity that partially undermines the trustlessness proposition; the party controlling the upgrade key effectively retains the power to alter the contract's behavior unilaterally.

2.2.4 Vulnerabilities and Security Challenges

The immutability of deployed smart contracts transforms coding errors from correctable mistakes into permanent, potentially catastrophic liabilities. Smart contract technology is still in its infancy, and bugs and vulnerabilities continue to be discovered in both newly deployed and legacy contracts. Several categories of vulnerability have been identified and extensively studied in the literature [56]. The reentrancy vulnerability is perhaps the most infamous. It occurs when a contract makes an external call to an untrusted contract before resolving its own state, allowing the callee to re-enter the caller's function and manipulate funds or state beyond the caller's intent. This vulnerability was exploited in the catastrophic 2016 attack on The DAO, resulting in the theft of approximately 60 million USD and the subsequent hard fork of the Ethereum network [16]. The integer overflow and underflow class of vulnerabilities arises from Solidity's fixed-size integer types; arithmetic operations that exceed the representable range silently wrap around, enabling attackers to manipulate balances or counters. The access control vulnerability occurs when functions that modify critical state variables lack appropriate visibility modifiers or ownership checks, allowing unauthorized parties to invoke privileged operations [57]. To assist developers in identifying these vulnerabilities, a rich ecosystem of automated analysis tools has emerged. Slither [19] is a widely adopted static analysis framework for Solidity contracts, capable of identifying a broad range of security issues including reentrancy, unchecked external calls, and incorrect visibility by analysing the contract's source code and abstract syntax tree without executing it. Manticore [21] employs symbolic execution to systematically explore contract execution paths, checking for violations of user-defined assertions. Mythril [20] uses a combination of symbolic analysis and taint tracking to identify security vulnerabilities. Despite the availability of these tools, their widespread adoption remains limited because they typically require local installation, command-line expertise, and are primarily intended for use during the final stages of development rather than as interactive aids during coding. Machine learning approaches to vulnerability detection have also gained traction. The Peculiar system [58] employs pre-trained models on crucial data flow graphs to detect reentrancy vulnerabilities, achieving a precision of 91.80% and a recall of 92.40% on a dataset of approximately 41,000 smart contract files, significantly outperforming conventional static analysers in recall. Such results suggest a promising complementary role for deep learning techniques in the security

auditing pipeline, although their sensitivity to out-of-distribution contract patterns and their relative opacity remain subjects of active investigation.

2.2.5 Development Paradigms and Accessibility Barriers

Smart contract development traditionally demands a profound understanding of both blockchain technology and specialized programming expertise. Developers must simultaneously master the semantics of the Solidity language, the operational model of the EVM (including gas cost optimization), security best practices, and the domain-specific logic they are trying to encode. This technical complexity creates a significant accessibility gap, limiting smart contract development to those with advanced software engineering skills while excluding the broader population of potential users who could benefit from blockchain technology [59, 33]. Several paradigms have been proposed to lower this barrier. Template-based approaches (TA) provide pre-built contract skeletons that users can instantiate by filling in parameters [60], but they are inherently limited to pre-anticipated use cases and lack the flexibility to handle novel contractual structures. Model-driven approaches (MDA) allow users to specify contract behaviour at a higher level of abstraction [61], using UML activity diagrams, BPMN models, or domain-specific languages, which are then automatically transformed into executable Solidity code. Visual-oriented approaches (VDA) enable contract construction through graphical drag-and-drop interfaces, making the development process accessible to users unfamiliar with programming syntax [62]. Tools such as the service presented by Kobusinska and Wierzbicki [52] enable contract creation by constructing diagrams from graphical blocks, which are then transformed into Solidity code. This approach aligns with the well-established programming paradigm of visual programming languages such as Google Blockly [1]. Wohrer and Zdun [63] proposed a domain-specific language with a clause grammar close to natural language, incorporating commonly occurring design patterns and automatically integrating them during code generation. Despite these efforts, all non-machine-learning or non-automated approaches still require users to possess some degree of technical literacy and familiarity with the target platform, a limitation that motivates the investigation of deep learning-based, and in particular Large Language Model-based, generation pipelines described later in this thesis.

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

2.3 Large Language Models

2.3.1 Foundation of Language Modelling and Large Language Models

The field of natural language processing (NLP) has undergone a paradigm shift over the past decade, moving from hand-crafted feature engineering and rule-based systems towards end-to-end neural approaches that learn representations directly from raw text. Early neural language models, introduced by Bengio et al. in 2003 [64], demonstrated that continuous-space word representations could substantially outperform n-gram statistical models on perplexity-based benchmarks. Recurrent Neural Networks (RNNs) [65] and Long Short-Term Memory (LSTM) networks [66] subsequently became the dominant architectures for sequential text processing, due to their ability to propagate information across variable-length sequences. The LSTM architecture, in particular, introduced gating mechanisms that effectively mitigate the vanishing gradient problem inherent to standard RNNs, enabling practical training of deep networks over long sequences. The watershed moment was the introduction of the Transformer architecture by Vaswani et al. [67]. Unlike RNNs and LSTMs, which process sequences token by token, the Transformer employs a self-attention mechanism that computes relationships between all pairs of tokens in a sequence simultaneously, enabling massively parallel training on modern GPU hardware and effectively capturing long-range dependencies in text. The encoder–decoder structure of the original Transformer, designed for machine translation, achieved state-of-the-art results on the WMT 2014 English-to-German and English-to-French benchmarks with substantially lower training cost than prior architectures. The paper has since accumulated more than 170,000 citations, making it among the most influential scientific publications of the twenty-first century [67]. Building on the Transformer, the BERT model (Bidirectional Encoder Representations from Transformers), introduced by Devlin et al. [68], proposed masked language modeling as a pre-training objective, enabling bidirectional context understanding by training a model to predict randomly masked tokens using both left and right contexts. BERT achieved state-of-the-art results on eleven NLP benchmarks, including the GLUE score of 80.5 and SQuAD v1.1 question answering F1 of 93.2, establishing pre-training followed by task-specific fine-tuning as the dominant paradigm for NLP. Simultaneously, the GPT (Generative Pre-trained Transformer) series from OpenAI

adopted a left-to-right (autoregressive) language modeling objective, training the model to predict the next token given all preceding tokens. A defining characteristic of modern AI research is the observation that model performance on a wide variety of tasks scales predictably with three factors: the number of model parameters, the volume of training data, and the amount of compute expended during training, the so-called scaling laws. GPT-3, introduced by Brown et al. in 2020 [69] with 175 billion parameters, demonstrated that scaling up language models greatly improves task-agnostic, few-shot performance, sometimes reaching competitiveness with prior state-of-the-art fine-tuning approaches, while requiring no gradient updates at inference time. This finding fundamentally altered the research community's understanding of what large neural networks are capable of, and inaugurated a period of rapid investment in ever-larger pre-trained models. Exploiting these scaling laws, researchers have trained increasingly large language models, ultimately giving rise to the category of Large Language Models (LLMs). These models, characterized by parameter counts ranging from billions to hundreds of billions, exhibit qualitatively new capabilities that smaller models do not possess, a phenomenon referred to as emergent abilities. These include multi-step reasoning, analogical reasoning, code generation, and instruction following. OpenAI's GPT-4, the latest iteration in the GPT series at the time of this writing, has exhibited superior capabilities in tasks related to natural sciences, engineering, and semantic understanding [25]. Other prominent LLMs include LLaMA 1 and 2 by Meta [70, 71], PaLM by Google [72], and Mistral by MistralAI [73]. Each of these models differs in training corpus, model architecture, context window length, licensing terms, and training paradigm which are factors that significantly influence their suitability for specific downstream tasks. A critical technique for aligning LLMs with human preferences and instructions is Reinforcement Learning from Human Feedback (RLHF). In this paradigm, a reward model is trained on human preference comparisons between model outputs, and the LLM is then fine-tuned using this reward signal via Proximal Policy Optimization (PPO). ChatGPT, released by OpenAI in November 2022, was trained as a common LLM and then, under reinforcement learning settings, learned from the ratings provided by humans on its responses, dramatically improving its ability to follow instructions and engage in coherent, helpful dialogue.

2.3.2 Prompt Engineering

Because LLMs are trained to continue text, the precise formulation of the input, the prompt, profoundly influences the quality and relevance of the model's output [74]. Prompt engineering refers to the practice of designing, optimizing, and iterating on input prompts to elicit desired model behaviours [75]. Unlike traditional software engineering, prompt engineering is inherently empirical: small changes in wording, ordering, or framing can produce dramatically different outputs. Several structured methodologies have been proposed to systematize prompt engineering. The CO-STAR framework structures prompts along six dimensions: Context, Objective, Style, Tone, Audience, and Response [38]. By explicitly specifying each of these dimensions, the CO-STAR methodology guides the model towards outputs that are contextually grounded, stylistically appropriate, and format-compliant. In the domain of smart contract generation specifically, Napoli et al. [37] demonstrated that CO-STAR-structured prompts produce more consistent and correct outputs compared to unstructured natural language requests. Advanced prompt engineering techniques include few-shot prompting, wherein the prompt includes a small number of input-output examples that implicitly demonstrate the desired task, and chain-of-thought (CoT) prompting [76], which encourages the model to reason step by step before producing a final answer. Wei et al. [76] demonstrated that CoT prompting substantially improves performance on complex reasoning tasks, particularly for arithmetic and logical problems that benefit from intermediate computational steps, with experiments showing that a 540-billion-parameter model guided by eight CoT exemplars achieves state-of-the-art accuracy on the GSM8K benchmark of math word problems.

2.3.3 LLMs in Software Enginering

The application of LLMs to software engineering tasks has attracted substantial research interest. LLM-powered tools have been evaluated across the software development life cycle, from requirements extraction and ambiguity resolution to code generation, test case synthesis, code review, translation between programming languages, and vulnerability detection and repair [77, 26]. Some developers report substantial productivity gains when using LLM-powered tools for routine coding tasks such as auto-completion and boilerplate generation; others argue that LLMs'

limitations, particularly their tendency to produce plausible-sounding but semantically incorrect code, a phenomenon sometimes called hallucination — make them better suited as assistive tools rather than autonomous software engineers [78, 79]. Sobania et al. [80] performed a comprehensive analysis of ChatGPT's automatic bug-fixing performance across a widely used benchmarking dataset. Their findings revealed that while ChatGPT could correct a meaningful fraction of bugs in isolation, its success rate declined substantially for bugs requiring understanding of broader programme context. Belzner et al. [77] surveyed LLM-assisted software engineering more broadly, identifying hallucination of non-existent APIs, inconsistent adherence to project-specific coding conventions, and difficulty handling large codebases as principal limitations. Despite these challenges, the same authors acknowledged that LLMs could function as valuable "pair programmers" when supervised by experienced developers. From a theoretical perspective, LLMs' facility with code generation can be attributed in part to the large volume of open-source code present in their training corpora. Platforms such as GitHub, Stack Overflow, and software documentation sites contribute billions of tokens of structured, syntactically valid code, giving LLMs extensive pattern exposure [81]. However, this training distribution differs substantially from the constraints of production code quality, security standards, and domain-specific correctness criteria, differences that become acutely apparent in high-stakes applications such as smart contract development.

2.4 Convergence: LLMs, Smart Contract, and the Research Agenda

The preceding sections have established the individual foundations of blockchain technology, smart contracts, and Large Language Models. This final section synthesizes these strands and articulates the specific research gaps that motivate the work presented in this thesis. The convergence of blockchain and artificial intelligence represents one of the most promising and underexplored frontiers in contemporary computer science [37].

Blockchain offers LLMs a domain in which the stakes of code correctness are exceptionally high since a single undiscovered vulnerability can result in the irreversible loss of millions of dollars, while LLMs offer blockchain a pathway to

dramatically lower the technical barrier for smart contract authorship. Understanding the precise conditions under which LLMs succeed and fail in this domain is therefore both scientifically significant and practically urgent.

Three interrelated gaps motivate the contributions of this thesis. First, prior evaluations of LLMs for smart contract generation have relied excessively on automated metrics, such as compilability rates and static analysis vulnerability counts that are demonstrably insufficient to capture semantic correctness. A more rigorous evaluation methodology, grounded in domain-specific design patterns and manual expert review, is needed. Second, the visualization of smart contract structure and behavior remains underdeveloped; tools that enable developers and non-developers alike to inspect, understand, and verify smart contracts before and after deployment could substantially mitigate the risks of deploying flawed code. Third, the specific failure modes of different LLMs in different contract domains have not been systematically characterized, leaving practitioners without actionable guidance on model selection and prompt design for specific use cases.

Chapter 3

Evaluating ChatGPT for Smart Contract Vulnerability Correction

3.1 Introduction

The rapid growth of the Ethereum ecosystem has been accompanied by a corresponding increase in the number of security exploits targeting deployed smart contracts. Because smart contract source code is publicly visible on the blockchain and the contracts themselves are immutable once deployed, even a single non-patched vulnerability can expose substantial funds to theft [82]. In response, the research community has developed a range of static analysis tools capable of systematically identifying well-known vulnerability classes in Solidity code, including Slither [19], Manticore [21], Mythril [83], SmartCheck [23], Securify [24], and Oyente [22]. Despite their diagnostic power, these tools are fundamentally detectors: they report the location and nature of a flaw, but the fix, hence rewriting the vulnerable code correctly, remains a manual task that demands deep familiarity with Solidity best practices and the Ethereum execution model.

The release of ChatGPT in November 2022 opened a new research direction. As described in Section 2.3, ChatGPT was trained as a general-purpose LLM and subsequently fine-tuned under reinforcement learning from human feedback (RLHF) to produce coherent,contextually grounded responses. Within months of its release, the model attracted widespread attention for its ability to generate, summarise, and debug code across a range of programming languages [80, 77]. At the time of this

study (April 2023), the most capable engine available to the general public was `gpt-3.5-turbo`.

Whether this capability could be directed at the problem of *correcting* security vulnerabilities in Solidity smart contracts had not been studied, and the question was treated as an exploratory one. It is important to note that an LLM predicts the most probable continuation of a token sequence conditioned on a given prompt: the model possesses no explicit knowledge of smart contract vulnerability patterns, no understanding of the EVM execution model, and no formal notion of program correctness. Therefore, any repair behavior emerges from statistical regularities in the training corpus rather than from principled security reasoning, placing an intrinsic ceiling on the reliability of purely LLM-driven correction and motivating the hybrid pipeline design described in Section 3.4.2.

1. An evaluation of ChatGPT's ability to process Solidity source code and return syntactically valid, compilable corrections.
2. A quantification of the vulnerability correction rate achieved by ChatGPT, both without any additional context (*code-only mode*) and with a structured bug description produced by Slither (*bug-description mode*).
3. An open-source, reusable pipeline that integrates ChatGPT and Slither to automate the detection–correction–verification cycle for Solidity smart contracts.

The remainder of the chapter is organized as follows. Section 3.2 provides the technical background. Section 3.3 surveys related work on the repair of LLM-based programs. Section 3.4 describes the experimental pipeline. Section 3.5 presents and analyzes the results. Section 3.6 discusses the implications. Section 3.7 concludes and outlines future work.

3.2 Background

3.2.1 Smart Contracts and the Solidity Language

The foundational architecture of the Ethereum platform, the semantics of the Solidity programming language, the EVM compilation and deployment pipeline, and the gas

model are described in detail in Sections 2.2.2 and 2.2.3 of Chapter 2. The present section highlights only the property most directly relevant to this chapter's research question: the immutability of deployed smart contracts.

Once a smart contract is transmitted to the Ethereum network and included in a confirmed block, its bytecode cannot be modified. This immutability guarantees consistent and trustless execution across all network participants, but it also means that vulnerabilities discovered after deployment cannot be patched in place [84]. Newer Solidity compiler versions have progressively addressed many historical weaknesses, but a large number of contracts compiled in older versions remain active on the live network and are potentially exploitable. Vulnerabilities typically arise from developers' unfamiliarity with Solidity semantics, from the reuse of third-party code containing flaws, or from the relative immaturity of the language [82]. It is this permanent, non-patchable nature of deployed code that motivates the investigation of automated *pre-deployment* correction approaches, such as the ChatGPT-based pipeline presented in this chapter.

3.2.2 Vulnerability Detection Tools and Selection of Slither

The principal static and symbolic analysis tools available for Solidity smart contracts, including Slither, Manticore, Mythril, SmartCheck, Securify, and Oyente, are surveyed in Section 2.2.4 of Chapter 2,together with their detection methodologies, maintenance status, and known limitations. This section justifies the choice of Slither [19] as the sole analysis tool in the pipeline of the present study.

Slither was selected over the available alternatives for three reasons specific to the requirements of an automated correction pipeline. First, among all publicly available smart-contract analysers, Slither has received the highest community endorsement, as evidenced by its GitHub star count; this makes it a broadly accepted proxy for correctness in the research community. Second, Slither operates through a lightweight intermediate representation (SlithIR) and completes a full vulnerability scan in approximately five seconds, which is essential for the iterative four-round query design described in Section 3.4.2. Third, unlike Manticore and Mythril, which rely on symbolic execution and require substantially more time to exhaustively explore all possible execution paths, Slither can analyse every contract in the dataset

within the time budget of an automated correction loop without sacrificing coverage of the vulnerability classes present in the SmartBugs-Curated benchmark [84].

3.3 Related Work

The broader landscape of LLM applications in software engineering, such as code generation, test synthesis, code review, and general vulnerability detection, is surveyed in Section 2.3.3 of Chapter 2, which also covers ChatGPT performance across domains such as education, scientific writing, and healthcare. The present section focuses exclusively on studies that evaluate LLMs in the context of automated program repair (APR), the closest prior art to the vulnerability correction task addressed in this chapter.

Sobania et al. [80] evaluated ChatGPT on the QuixBugs benchmark, consisting of a dataset of 40 short Python and Java programs each containing a single-line bug, and found that the model successfully corrected 19 out of 40 instances without any additional context. When the conversational capability of ChatGPT was exploited by providing a hint in a follow-up turn, the success rate increased to 31 out of 40, outperforming state-of-the-art baselines such as CoCoNuT and Codex. The authors highlighted that ChatGPT's dialogue-based interaction represents a qualitatively different mode of use compared with traditional APR tools: the developer can provide incremental guidance without reformulating the entire prompt. Prenner et al. [85] reached similar conclusions when evaluating OpenAI's Codex on the same benchmark, finding a comparable fix rate and noting that the model performed better on Python than on Java. Both studies observed that providing additional context about the nature of the bug substantially increased success rates.

A complementary direction has explored hybrid pipelines that combine LLMs with static analysers. Jin et al. [86] proposed InferFix, a framework that pairs the Infer static analyser with a fine-tuned 12-billion-parameter Codex model to detect and repair critical security and performance bugs in Java and C# code. By augmenting prompts with bug type annotations and semantically similar fixes retrieved from an external memory store, InferFix achieved a top-1 repair accuracy of 65.6% on C# and 76.8% on Java. These results demonstrate that structured, tool-derived context can meaningfully improve the repair fidelity of LLMs, a finding that directly motivates the two-prompt design adopted in the present study.

Despite this breadth of evaluation, no prior work had examined whether ChatGPT could correct security vulnerabilities in Ethereum smart contracts specifically. The smart contract setting differs from general APR benchmarks in several important respects: Solidity is substantially less represented in LLM training corpora than Python or Java; the vulnerability classes are tied to the specifics of the EVM execution model; and the absence of a test suite means that correctness must be assessed by a static analyzer rather than by execution. The present chapter fills this gap by applying ChatGPT to the SmartBugs-Curated benchmark [84] and using Slither as a fully automated vulnerability detection tool, enabling a controlled, reproducible evaluation that requires no manual code inspection.

3.4 Proposed Approach

3.4.1 Dataset

The study relies on the *SmartBugs-Curated* dataset [84], a benchmark comprising 143 vulnerable Solidity smart contracts whose vulnerabilities have been manually annotated according to the Decentralized Application Security Project (DASP) classification. The manual annotations are not used directly in this study. Instead, Slither serves as the ground-truth oracle in both the pre-correction and post-correction phases, in accordance with the objective of evaluating a *fully automated* vulnerability management pipeline that does not require expert human judgment at runtime.

3.4.2 Pipeline Design

The experimental pipeline consists of four sequential steps.

Step 1: Code Cleaner. Each smart contract's source code is pre-processed to remove all Solidity comments both single-line (`//`) and multi-line(`/* ... */`) blocks using a regular-expression filter. This step minimizes the token count of each submitted prompt, which is critical given that the `gpt-3.5-turbo` model was subject to a 4,096-token limit at the time of the study.

Step 2: Vulnerability Finder (pre-correction). The cleaned source code is passed to Slither, which produces both the count and the natural-language description

of each detected vulnerability, together with its severity level (High, Medium, or Low) and the line number at which it was identified. This step serves two purposes: (i) it records the reference vulnerability profile against which subsequent corrections are evaluated, and (ii) it generates the structured error list that can optionally be appended to the ChatGPT prompt. The user may additionally filter the Slither output by severity level; when no filter is specified, only High-severity vulnerabilities are communicated to ChatGPT by default.

Step 3: ChatGPT Questioner. A prompt is constructed from the cleaned source code and submitted to ChatGPT via the OpenAI API using the `gpt-3.5-turbo` engine. Two prompt schemas are employed in parallel:

- *Code-only mode.* The prompt contains only the cleaned source code, preceded by the instruction: *"Fix the vulnerability in the Solidity code. Provide the fixed contract."* `<source code>`
- *Bug-description mode.* The Slither error list is appended to the code-only prompt, enumerating each detected vulnerability by type and line number: *"Fix the vulnerability in the Solidity code. Provide the fixed contract.* `<source code>` *Errors:* `<error 1> <error 2>...`*"*

Following the approach proposed by Sobania et al. [80], each prompt is submitted four times independently, with a one-second offset between successive calls to avoid rate-limit errors. If a response fails after five attempts (due to server overload or token limit violations), the contract is skipped. All returned code is saved for subsequent verification.

Step 4: Vulnerability Finder (post-correction). The patched contract returned by ChatGPT is re-analysed by Slither. A correction is judged successful if the patched code (i) introduces no new vulnerabilities and (ii) contains fewer vulnerabilities of at least one severity level compared with the original contract.

3.4.3 Best-Round Selection Criterion

Since each contract is queried four times, the four resulting patches are not aggregated but compared, and a single *best round* is identified per contract. The selection

proceeds in two steps. First, a round is declared *valid* only if the patched code introduces no new vulnerabilities relative to the original; rounds that regress the contract's security profile are discarded regardless of how many existing vulnerabilities they remove. Second, among the remaining valid rounds, the one that eliminates the largest total number of vulnerabilities is selected, with ties resolved by prioritising reductions at the highest severity level first, i.e. High, then Medium, then Low.

3.5 Results

3.5.1 Overall Correction Rates

Three smart contracts exceeded the 4,096-token limit in code-only mode and eight exceeded it in bug-description mode; these were excluded from the respective analyses, yielding effective sample sizes of 140 contracts (560 API calls) and 135 contracts (540 API calls). Of the 560 calls in code-only mode, 56 returned non-compilable output; in bug-description mode, 57 calls produced code that Slither could not compile.

Table 3.1 reports the aggregate correction results. Across all four rounds, ChatGPT fixed at least one vulnerability in 52.8% of the processed contracts in code-only mode and 53.1% in bug-description mode. Selecting the best round for each contract raised the correction rate to 57.1% in code-only mode and 58.5% in bug-description mode, which is a $+1.4$ percentage-point improvement attributable to the structured Slither hint. ChatGPT resolved *all* vulnerabilities detected by Slither in at least one round for 27.1% of contracts in code-only mode and 26.7% in bug-description mode.

Table 3.1 Overall correction results in code-only and bug-description modes.

Metric	**Code-only**	**Bug description**
Average across all rounds	52.8%	53.1%
Best round	57.1%	58.5%
All vulnerabilities fixed	27.1%	26.7%

3.5.2 Results by Vulnerability Severity

Tables 3.2 and 3.3 show the correction and introduction rates by severity level in all four rounds and the best-round selection.

In **code-only mode** (Table 3.2), the source dataset across 140 processed contracts contains 130 High-severity, 115 Medium-severity, and 169 Low-severity vulnerabilities. In the best-round, ChatGPT corrected 49% of High-severity, 70% of Medium-severity, and 45% of Low-severity vulnerabilities, while introducing 6, 9, and 5 new vulnerabilities respectively. The markedly higher success rate on Medium-severity issues is stable across all four rounds and suggests that the patterns associated with medium-severity Solidity flaws are better represented in the model's pre-training data, or that they correspond to simpler and more stereotyped code transformations.

In **bug-description mode** (Table 3.3), the best-round figures are closely comparable: 49% of High-severity, 72% of Medium-severity, and 50% of Low-severity vulnerabilities are corrected, with slightly fewer newly introduced vulnerabilities in the High and Low severity categories. The reduction in regressions when bug descriptions are provided is consistent with the hypothesis that the Slither error list enables ChatGPT to focus its edits on the reported locations, thereby limiting inadvertent changes elsewhere in the contract.

Table 3.2 Fixed and added vulnerabilities in code-only mode over 140 processed contracts. Percentages computed over source counts.

	Source	Round 1		Round 2		Round 3		Round 4		Best	
Severity	**Vuln.**	**Fix**	**Add**	**Fix**	**Add**	**Fix**	**Add**	**Fix**	**Add**	**Fix**	**Add**
High	130	61 (44%)	9	61 (44%)	8	62 (47%)	7	62 (47%)	6	64 (46%)	6
Medium	115	77 (67%)	12	78 (68%)	11	79 (69%)	9	79 (69%)	10	81 (70%)	9
Low	169	69 (41%)	5	69 (41%)	4	73 (43%)	5	73 (43%)	4	76 (45%)	5

Table 3.3 Fixed and added vulnerabilities in bug-description mode over 135 processed contracts. Percentages computed over source counts.

	Source	Round 1		Round 2		Round 3		Round 4		Best	
Severity	**Vuln.**	**Fix**	**Add**	**Fix**	**Add**	**Fix**	**Add**	**Fix**	**Add**	**Fix**	**Add**
High	125	57 (46%)	9	57 (46%)	7	59 (47%)	7	59 (47%)	6	62 (49%)	5
Medium	111	76 (68%)	11	75 (68%)	10	78 (70%)	8	78 (70%)	10	80 (72%)	8
Low	162	72 (44%)	4	67 (41%)	3	78 (48%)	4	77 (47%)	3	82 (50%)	4

3.6 Discussion

3.6.1 Compilability and Output Reliability

An important secondary finding concerns the compilability of ChatGPT's output. Taken into account the overall 1100 API call made in code-only mode and bug description mode, 89.7% of all API responses returned code that Slither was able to compile, which is a necessary precondition for any production-ready vulnerability-management pipeline. This result indicates that ChatGPT consistently preserves the syntactic structure of Solidity contracts even when it fails to eliminate the target vulnerability, and that the proportion of cases in which the model's edits introduce structural corruption is relatively modest (approximately 10%). This baseline compilability distinguishes ChatGPT from naive code-generation approaches and confirms that it can be integrated into automated pipelines without a high risk of invalidating the entire contract.

3.6.2 Effect of Providing Structured Bug Descriptions

The $+1.4$ percentage-point improvement observed when Slither's structured error descriptions are included in the prompt is modest but directionally consistent across all four rounds and both correction metrics. Two factors may explain the limited magnitude of this effect. First, Slither's error messages are terse and tool-specific: they communicate vulnerability type and line location in compact form but do not articulate the underlying logical flaw in natural language, which would provide a richer signal to the model. Second, the 4,096-token limit constrained the usable prompt size in bug-description mode, causing eight additional contracts to be excluded compared to code-only mode. Future work could test more descriptive, natural-language

reformulations of static analyser warnings and could benefit from the substantially larger context windows available in newer model versions.

3.6.3 Limitations

Several limitations constrain the generalization of the conclusion of this study. First, Slither is used as the sole correctness oracle. As a static analyser, Slither can neither detect all classes of Solidity vulnerabilities nor guarantee the *semantic* correctness of a corrected contract; a contract may pass Slither's checks yet still contain logical flaws invisible to pattern-based detectors. Second,the study evaluates only a single-turn interaction, without exploiting the conversational capabilities of ChatGPT, for instance, by iterating on a failed correction in a follow-up turn. Prior work on general program repair has shown that such dialogue-based iteration can substantially raise success rates [80]. Third, the evaluation is tied to GPT-3.5 Turbo as it existed in April 2023; subsequent model versions, including GPT-4 and more recent releases, may produce substantially different results on the same benchmark.

3.6.4 Implications for Automated Vulnerability Management

Despite these limitations, the results establish that ChatGPT could function as a viable first-pass patch generator within an automated vulnerability-management pipeline. A best-round correction rate of 57.1% across 140 contracts, achieved with an average of four API calls per contract, compares favorably with the manual effort typically required to interpret a static analysis report and implement a correct fix. The open-source implementation of the pipeline enables its integration into continuous integration (CI) environments as a post-commit stage, where Slither detects vulnerabilities in newly committed Solidity code, ChatGPT proposes candidate patches, and a second Slither run validates each patch before surfacing it to the developer. This design minimizes false positives in correction suggestions and reduces the cognitive burden on the developer to the final review step.

The study also surfaces a structural insight that motivates the subsequent chapters of this thesis: automated evaluation metrics, in this case, the binary verdict of Slither, have inherent limits. Slither cannot assess whether a corrected contract preserves the original business logic, introduces subtle semantic regressions, or produces

code that would be considered idiomatic and secure by an experienced Solidity developer. This motivates the adoption of a complementary expert-driven evaluation methodology for the more ambitious task of *generating* complete smart contracts from natural-language specifications, as developed in Chapters 5and 6.

3.7 Conclusions

This chapter presented the first empirical study of ChatGPT's capacity to correct security vulnerabilities in Ethereum smart contracts, leveraging the SmartBugs-Curated benchmark [84] and the Slither static analyser [19] as vulnerability ground truth. The main findings are as follows.

ChatGPT returned compilable Solidity code in 89.7% of cases, demonstrating reliable syntactic fidelity. In the best of four attempts, it fixed at least one Slither-detected vulnerability in 57.1% of the processed contracts when provided with code only, and in 58.5% of contracts when a structured bug description was also provided, a $+1.4$ percentage-point improvement attributable to the additional context. The model was most effective on Medium-severity vulnerabilities (best-round fix rate: 70–72%) and least effective on Low-severity ones (45–50%), with High-severity vulnerabilities corrected at an intermediate rate of approximately 49%.

These findings demonstrate that an LLM can act as a cost-effective automated patch generator for smart contract vulnerabilities when integrated with a static analyser in a structured pipeline. At the same time, they reveal the ceiling imposed by purely automated evaluation: Slither's binary verdict provides no information about the semantic correctness or practical quality of the generated code. Addressing this limitation, by introducing expert manual evaluation frameworks capable of assessing logical soundness, security, and correctness, is the central challenge taken up in the chapters that follow.

Chapter 4

Leveraging Large Language Models for Automatic Smart Contract Generation

4.1 Introduction

The challenge of automating the translation of natural language agreements and legal documents into executable smart contracts represents one of the most consequential open problems in applied blockchain research. As established in Chapter 1, the population of individuals and organizations that could derive tangible benefit from smart contract technology is substantially larger than the population currently capable of writing the correct Solidity code. The preceding chapter demonstrated that Large Language Models can be directed at the narrower problem of *correcting* pre-existing vulnerable contracts, achieving meaningful success rates when coupled with a static analyzer. The natural extension of that line of inquiry is whether the same class of models can be directed at the significantly more ambitious task of *generating* complete, compilable, and reasonably secure smart contracts from scratch, starting from a legal agreement or a plain-language textual description.

This chapter presents the design, implementation, and experimental evaluation of a pipeline for automatic smart contract generation based on OpenAI's `gpt-4-0125-preview` model [87]. The pipeline accepts a legal agreement or textual description as input and produces a Solidity smart contract, subjecting the output

to automated quality assessment through a multi-metric benchmarking suite. The evaluation is conducted on a property rental agreement sourced from an online legal document platform, chosen because it incorporates diverse numerical values, textual descriptions, and address-type parameters, thus providing a comprehensive test of the model's ability to extract and translate heterogeneous contractual information into Solidity constructs.

A central concern motivating the study is the role of *prompt engineering* in determining output quality. Rather than treating the prompt as a fixed design choice, the research constructs a graded series of 17 prompt templates around the CO-STAR framework [38], an acronym denoting six design dimensions: Context, Objective, Style, Tone, Audience, and Response. From this initial pool, the six highest-scoring and most representative templates are retained for detailed analysis, arranged so that consecutive templates differ in a single component. A complementary investigation examines the effect of the model's temperature parameter on the quality, consistency, and reproducibility of the generated contracts.

The chapter makes the following contributions. It proposes an end-to-end pipeline for translating legal agreements and textual descriptions into Solidity smart contracts via a single LLM interaction, without requiring domain-specific templates or intermediate formal representations. It introduces a multi-metric benchmarking suite that evaluates the generated contracts along seven quantifiable dimensions, combined into a Weighted Total Score. It provides empirical evidence that compilability of GPT-4-generated Solidity contracts is achievable at rates as high as 98.1%, and it reports an exploratory analysis of the temperature parameter in which no systematic trend in mean Weighted Total Score was observed across the range $[0, 1]$. Finally, it demonstrates that structured CO-STAR prompts significantly and consistently outperform unstructured prompts, with prompts that specify a senior Solidity developer and include explicit software requirements yielding the highest overall quality.

The remainder of this chapter is organized as follows. Section 4.2 reviews the previous work on smart contract generation. Section 4.3 describes the architecture of the system and the selection of the component. Section 4.4 presents the experimental design, including prompt formulation, temperature analysis, and consistency evaluation. Section 4.5 reports and discusses the results. Section 4.6 concludes and identifies directions for future work.

4.2 Related Work

Automatic or semi-automatic generation of smart contracts has attracted growing research attention over the past several years, with methodological approaches falling into four broad categories: template-based methods, natural language processing approaches, domain-specific languages, and deep learning techniques.

Template-based methods were among the first responses to the accessibility barrier described in Section 2.2.5. Roche et al. [88] proposed the Ergo programming language as a tool to encode legal contracts in a formally specified executable format. The Accord Project [89] elaborated on this direction by developing a framework that includes 56 templates for common legal agreements alongside a natural language layer to write business logic. While the Accord Project significantly reduces the expertise required for contract authoring, it remains a template-bound approach: the set of contractual structures it can represent is determined by the pre-existing template library, and agreements that fall outside this inventory require human intervention for new templates.

Natural language processing approaches aim to reduce template dependency by extracting structured information directly from legal text. Fang et al. [90] proposed iSyn, a semi-automatic tool that extracts contract terms from legal agreements and maps them to Solidity code fragments. Although its results are notable, iSyn operates as a template-based tool internally, limiting its expressiveness to payments, purchases, file uploads, and time-constrained actions; broader contractual structures involving late fees, deposits, and complex conditional logic are not considered.

Domain-specific language approaches avoid the limitations of natural language ambiguity by requiring the contract specifier to express business logic in a structured, machine-readable format before code generation. Regnath and Steinhorst [91] proposed SmaCoNat, a DSL that expresses smart contract behaviour in natural language syntax while retaining formal semantics. Choudhury et al. [92] employed domain-specific ontologies and semantic rules to build a contract template that is automatically translated into smart contract code through Go scripts. Hamdaqa et al. [93] proposed iContractML 2.0, a domain-specific language supporting multiple blockchain platforms that models contracts using Participant, Transaction, Asset, Relationship, Condition, and Event classes. Chen et al. [94] advanced this direction with SPESC, a domain-specific language (DSL) targeting Ethereum that is designed

to be populated by NLP-extracted terms from text documents, though their system still requires human intervention to generate the final contract code. Wohrer and Zdun [63] devised the Contract Modelling Language (CML), a linguistic framework designed to describe smart contract elements and guide their generation through clause grammar, although its requirement for familiarity with CML syntax constrains its appeal among non-technical users. While DSL-based approaches generally produce predictable and formally sound outputs, they impose the burden of learning the DSL on the contract author, replacing one form of technical expertise with another.

Deep learning approaches represent the most recent and most flexible generation paradigm. Hao et al. [95] proposed a system that generates contract code templates by applying a K-means++ classifier to labeled smart contracts, ensuring that the generated template captures the basic and secure functions appropriate to the contract type. However, with the widespread availability of powerful pre-trained LLMs, the DSL and deep learning paradigms began to converge toward a more direct text-to-code approach. Qasse et al. [96] proposed Chat2Code, a chatbot-based approach that uses an LLM to automatically generate smart contract source code through a conversational interface, achieving promising results but requiring ongoing dialog between the user and the chatbot throughout the generation process. Petrovic and Al-Azzoni [97] presented a model-driven framework that uses ChatGPT to automate smart contract generation, treating the task as a dialog between a user-provided iContractML model and the ChatGPT service; the approach demonstrates flexibility relative to template-based methods, but necessitates the user's familiarity with the iContractML formalism as a prerequisite.

The approach presented in this chapter differs from the previous work in three key respects. First, it operates through a single LLM interaction rather than requiring an ongoing dialog with the model or a pre-existing formal specification of the contract logic. Second, it introduces a structured benchmarking suite with quantifiable metrics, enabling a rigorous and reproducible assessment of the quality of the generated contract. Third, and unique in this literature, it systematically investigates the effect of prompt engineering and temperature settings on output quality across multiple evaluation dimensions, providing the research community with empirical guidance on how to optimize LLM-based contract generation.

4.3 System Architecture and Component Selection

The objective of the proposed system is to develop an autonomous pipeline capable of translating legal documents and textual descriptions into secure Solidity smart contracts. The pipeline comprises three pivotal components: an LLM responsible for generating contract source code from a natural language input, a security analysis tool for vulnerability detection and scoring, and a comprehensive benchmarking suite to assess the quality of the output.

4.3.1 LLM Selection

To select the LLM component, the evaluation criteria included performance on tasks related to natural sciences and engineering, semantic understanding, accessibility, licensing terms, context window size, and documented academic validation in the area of smart contract generation. Based on the survey by Chang et al. [25], which benchmarks a wide range of LLMs across multiple task categories, OpenAI's `gpt-4-0125-preview` was identified as exhibiting superior capabilities in the Natural Sciences and Engineering cluster and in semantic understanding tasks. This distinction is particularly relevant for translating legal contracts that require simultaneous comprehension of technical engineering constructs and context-dependent legal terminology. The widespread availability of GPT-4's training data from GitHub, which hosts a large volume of Solidity repositories, further supports the expectation of high code generation quality. The model was accessed through the OpenAI API at the `gpt-4-0125-preview` checkpoint, the most advanced publicly available GPT-4 iteration at the time of the study.

4.3.2 Security Analysis: Selection of Slither

The security analysis component is responsible for evaluating the generated contracts for known vulnerability classes, providing both a binary compilability verdict and a quantified vulnerability score. As discussed in Section 3.2.2, Slither [19] was selected over available alternatives for its comprehensive detection capabilities, its ability to analyze contracts at the Solidity source code level, its fast execution time, and its integration with the broader Trail of Bits toolchain. Its widespread adoption

Table 4.1 Summary of smart contract Evaluation Metrics

Metric	Evaluation Criteria/Weight
Compilability	1 if compiles, 0 if does not
Vulnerability Assessment	-1 (high), -0.5 (medium), -0.25 (low)
Solidity Version	1 if $\geq 0.8.0$, 0 if $< 0.8.0$
Number of Contracts Generated	0.25 per contract
Number of Functions	0.15 per function
Presence of Comments	1 if present, 0 if absent
Number of Initialized Parameters	0.5 per parameter

in related smart contract generation and repair studies [98, 36] further supports its use as a proxy for correctness in automated evaluation pipelines.

4.3.3 Target Blockchain and Language

Among Turing-complete blockchain platforms, Solidity on the Ethereum Virtual Machine was selected as the target language. This choice is justified by Solidity's status as the most widely used language for smart contract development and by the density of open-source Solidity repositories on GitHub, which constitute a significant fraction of GPT-4's training data. Importantly, Solidity contracts compiled to EVM bytecode can be deployed not only on the Ethereum mainnet but also on Layer-2 solutions such as Optimism and Arbitrum, as well as on other EVM-compatible chains including Binance Smart Chain and various Cosmos network zones, making the approach broadly applicable across the EVM ecosystem.

4.3.4 Evaluation Metrics

Smart contracts require a thorough evaluation to ensure reliability, security, and effectiveness. Informed by the framework of software metrics for smart contracts proposed by Tonelli et al. [99], the benchmarking suite incorporates seven quantifiable metrics, each assigned a weight reflecting its importance in the context of automated contract generation. Table 4.1 summarises the metrics and their scoring criteria, which are described individually below.

Compilability is a binary metric scoring 1 if the generated Solidity code compiles without errors and 0 otherwise. It constitutes a necessary precondition for any subsequent analysis and is the most fundamental quality indicator.

Vulnerability Assessment quantifies the security quality of the generated code by classifying each vulnerability detected by Slither as High, Medium, or Low severity, with corresponding penalty weights of -1, -0.5, and -0.25 respectively.

Solidity Version scores 1 if the contract specifies a pragma version $\geq 0.8.0$, and 0 otherwise. Contracts compiled with Solidity 0.8.0 or later benefit from built-in arithmetic overflow and underflow checks, automatically mitigating one of the most historically prevalent vulnerability classes.

Number of Contracts Generated awards 0.25 per contract present in the output. This metric captures the model's propensity to modularize complex contractual logic across multiple interacting contracts, which is an indicator of architectural sophistication.

Number of Functions awards 0.15 per function defined in the contract, reflecting the degree of functional decomposition and code organization.

Presence of Comments is a binary metric scoring 1 if the generated code includes inline documentation and 0 otherwise. Code comments are essential for maintainability and provide evidence of the model's ability to self-document its generated logic.

Number of Initialized Parameters awards 0.5 per initialized parameter. This metric assesses whether the model correctly extracts concrete values from the input agreement (numerical rents, addresses, dates) and instantiates them as initialized state variables or constructor arguments, rather than leaving them as unset placeholders.

The aggregate of all weighted metric scores constitutes the *Weighted Total Score* (WTS), which serves as the primary composite quality indicator to compare prompts and temperature settings.

4.4 Experimental Design

The experimental evaluation follows a three-stage procedure. The first stage assesses the effect of the temperature parameter on the quality of the output. The second

stage evaluates the consistency and stability of the LLM's response under fixed temperature settings across repeated invocations of the same prompt. The third stage determines which prompt formulation yields the highest quality outputs, as measured by the WTS.

4.4.1 Test Document

A property rental agreement obtained from an online legal document provider served as the primary test input throughout all experiments. As previously mentioned, this type of agreement was selected because it incorporates diverse parameter types, such as numerical rent values, textual address fields, date ranges, and conditional clauses governing deposits, late fees, and termination, thereby providing a comprehensive stress test of the model's information extraction and code generation capabilities.

4.4.2 Prompt Formulation

Contemporary prompt engineering practice predominantly employs chain-of-thought [76] and related stepwise reasoning techniques. However, these methods presuppose an interactive user-model dialogue, which is incompatible with the single-interaction design of the proposed pipeline. Similarly, few-shot and one-shot prompting [69] were excluded because they require exemplary smart contract instances to guide the model's output, while providing a pre-existing example would undermine the generativity of the approach. Given these constraints, the CO-STAR framework [38], which is a systematic prompt construction methodology developed by GovTech Singapore, was adopted as the principled approach to prompt design. CO-STAR structures prompt construction along six dimensions: *Context* (C), which provides the LLM with relevant background information about the use case; *Objective* (O), which articulates the specific task; *Style* (S), which specifies the preferred writing register; *Tone* (T), which calibrates the affective register of the response; *Audience* (A), which identifies the intended recipient of the output; and *Response* (R), which specifies the desired output format, in this case Solidity source code.

Not all six dimensions were instantiated as separate clauses, and two clauses that CO-STAR does not name were added. Context fixes the persona the model is asked to adopt; Objective states the generation task; Style selects the register

Table 4.2 Components of the six prompt templates retained for the experiments. Each row gives a component, the literal clause instantiating it, and the templates in which that clause occurs. *Requirements* and *Incentive* are auxiliary clauses rather than CO-STAR dimensions; *Tone* and *Response* are not realised as standalone clauses. The full text of the six templates is reproduced in Appendix A.

Component	**Clause used**	**Templates**
Context (C)	You are a senior Solidity developer.	PR12–PR17
Incentive (auxiliary)	*omitted*	PR12–PR15
	A company will pay you $500,000 for the completion of the project.	PR16, PR17
Objective (O)	Write a smart contract in Solidity based on the following legal agreement	PR12–PR14
	Write a smart contract in Solidity that reflects payments, conditions, constraint, and termination of the following legal agreement	PR15–PR17
Requirements (auxiliary)	*omitted*	PR12, PR13
	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.	PR14–PR17
Style (S)	Be creative and use your own style.	PR12
	Be professional and use a formal style.	PR13–PR17
Audience (A)	You will deliver the smart contract to another senior developer.	PR12–PR16
	You will deliver the smart contract to your company's legal department.	PR17
Tone (T)	*not instantiated*: no separate tone clause is used, the register being carried by the Style clause.	—
Response (R)	*not instantiated*: the output format, Solidity source code, is fixed by the Objective clause and further constrained by the Requirements block.	—

of the response; and Audience names the intended recipient of the contract. Tone was not instantiated as a standalone clause, the register being carried by the Style clause, and Response was not instantiated either, since the required output format is already fixed by the Objective clause, which asks explicitly for a smart contract in Solidity. Two auxiliary clauses were added to the templates: a *Requirements* block enumerating the technical constraints the generated code must satisfy, which operationalises part of the Response dimension, and an *Incentive* clause stating a nominal payment for completion of the task, included in order to test whether such framing affects output quality. Table 4.2 lists every component that occurs in the prompts used in the experiments, together with the literal clause instantiating it.

The 17 candidate templates were not produced by exhaustively enumerating this component space. The values considered during prompt design, namely three Context personas (a Master's student in Computer Science, a junior developer, and a senior Solidity developer), three Objective variants including omission, three Style variants including omission, four Audience recipients (a professor, a supervisor, a peer senior developer, and the company's legal department), and the presence or absence of the Requirements and Incentive clauses, span a combinatorial space of 432 templates, which the cost of the GPT-4 API at the time of the study made impractical to explore in full. Instead, 17 templates, PR1 to PR17, were written by hand as a graded series in which the degree of specification increases from bare instructions to fully constrained ones. All 17 were exercised in a preliminary screening phase, and the six highest-scoring and most representative templates, PR12 to PR17, were retained for the temperature and consistency experiments reported below. The retained six all fix the Context at a senior Solidity developer and address either a peer developer or a legal department; the Master's student and junior developer personas, and the professor and supervisor recipients, appear only among the discarded candidates and therefore play no part in the results of Section 4.5.

The six retained templates form a nested series in which each template differs from its predecessor in exactly one component, so that the contribution of a component can be read off the corresponding pair. PR12 and PR13 differ only in Style, creative against formal. PR13 and PR14 differ only in the presence of the Requirements block. PR14 and PR15 differ only in the specificity of the Objective, which in PR15 names payments, conditions, constraints, and termination as the aspects of the agreement that the code must reflect. PR15 and PR16 differ only in the presence of the Incentive clause. PR16 and PR17 differ only in the Audience, which moves from

a peer developer to the company's legal department. It is this one-factor-at-a-time arrangement, rather than the size of the candidate pool, that allows the effect of individual components on the WTS to be isolated in Section 4.5.3.

The two extremes of the series illustrate the construction. Prompt *PR12*, the least specified of the six, combines the Context, a generic Objective, a creative Style, and a peer-developer Audience:

> 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.

Prompt *PR17*, the most specified of the six, adds the Incentive clause, the detailed Objective, and the Requirements block, and redirects the Audience to the company's legal department:

> 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.

The complete set of 17 candidate templates is reproduced in full in Appendix A, comprising the four remaining retained templates PR13 to PR16 and the eleven candidates discarded after the screening phase; the script that composes them from their CO-STAR components is available in the project repository[1].

4.4.3 Temperature Analysis

The temperature parameter of a language model governs the degree of stochasticity in token sampling. At temperature 0, the model deterministically selects the highest-probability token at each step, producing maximally reproducible outputs. At temperature 1, sampling is performed according to the full probability distribution, introducing creative but potentially inconsistent variation. Values greater than 1, up to the API maximum of 2, further amplify stochasticity. In the present study, each of the six selected prompts was evaluated at five temperature values: $\{0.0, 0.2, 0.5, 0.7, 1.0\}$. Temperature values greater than 1.0 were excluded because preliminary tests found that the output in this regime was not consistently viable, typically producing incoherent or structurally malformed code. The WTS was computed for each prompt-temperature pair to quantify the effect of temperature on contract quality.

4.4.4 Consistency Evaluation

Following the approach of Sobania et al. [80] to evaluate LLM consistency in automated program tasks, the consistency phase subjected each of the six selected prompts to four independent API invocations at a fixed temperature of 0.5. This temperature value was selected as an intermediate point between the extremes of determinism and maximum stochasticity, consistent with standard practice in LLM-based code generation experiments. Quadruple repetition allows for the assessment of the variance in WTS across identical invocations, providing a direct empirical measure of the reliability and predictability of the pipeline under production conditions.

[1] https://github.com/BChain4all/pipeline

4.5 Results and Discussion

4.5.1 Compilability Rate

Out of the 54 API requests made to the GPT-4 endpoint, including 4 repetitions for each of the 6 selected prompts and one request per temperature value per prompt, the model produced compilable Solidity code in 98.1% of cases. This result is substantially higher than the compilability rate of 89.7% observed for GPT-3.5 Turbo (+8.4 percentage points) in the vulnerability correction study presented in Chapter 3, and reflects both the greater code generation capability of GPT-4 relative to GPT-3.5 and the richer specification provided by the input of the legal agreement. The high compilability rate indicates that the model consistently interprets legal language as a source of constructable contractual logic and produces syntactically and structurally valid Solidity code in the overwhelming majority of invocations. Given that OpenAI models are trained on extensive GitHub corpora that include large volumes of Solidity code, this high success rate is consistent with theoretical expectations about the quality of the training distribution for EVM-oriented code generation tasks.

4.5.2 Effect of the Temperature Parameter

Contrary to the initial hypothesis that higher temperature settings would degrade contract quality by introducing greater lexical and structural stochasticity,the analysis revealed that variations in the temperature parameter did not produce a systematic trend in mean WTS across the range $[0.0, 1.0]$. As illustrated by the results in Figure 4.1, the WTS curves remain largely flat and overlap throughout the temperature spectrum, with differences between consecutive temperature settings generally falling within the variance expected from random sampling noise. This stability is a consequential finding for practitioners: it suggests that the LLM pipeline can be deployed across a broad range of temperature settings without performance degradation, eliminating the need for temperature calibration as a prerequisite for deployment. It also suggests that the model's internal representation of Solidity syntax and contractual structure is sufficiently robust to withstand moderate levels of sampling stochasticity.

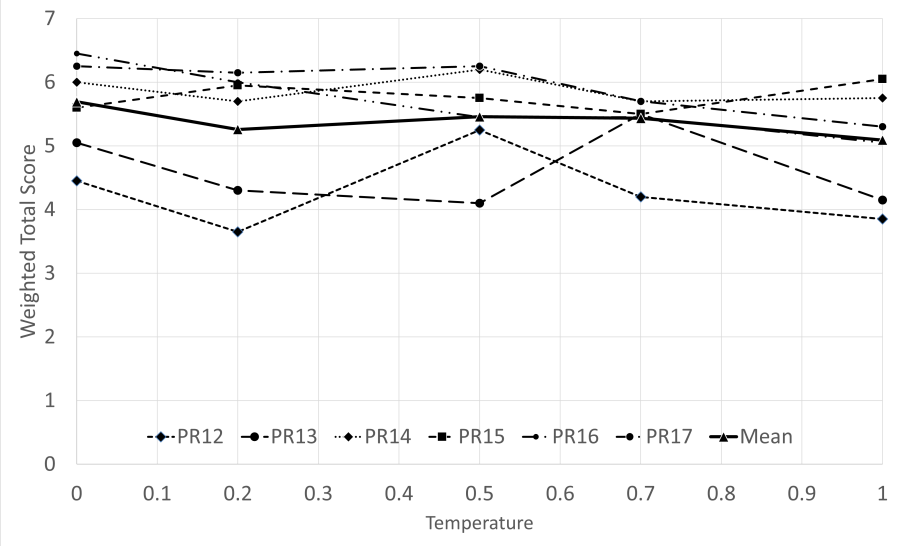


Fig. 4.1 Temperature effect on Weighted Total Score computed on the automatically generated smart contracts.

4.5.3 Prompt Quality and Consistency

The consistency analysis produced the most practically significant findings of the study. The four independent invocations of prompts PR15, PR16, and PR17, the three templates that pair the detailed Objective with the Requirements block, yielded consistently high and low-variance WTS values, with interquartile ranges approaching zero across the four repetitions as Figure 4.2 shows. This contrasts sharply with the behavior of prompts PR12, PR13, and PR14, which all retain the generic Objective and which, in the case of PR12 and PR13, also omit the Requirements block. Prompt PR14, which supplies the Requirements block but retains the generic Objective, produces the lowest mean WTS of the six templates and the only invocation scoring close to zero, suggesting that the technical constraints alone do not compensate for an Objective that leaves the aspects of the agreement to be encoded unstated, and that underspecified prompts expose the stochastic dimensions of the model's generation process, leading to unpredictable output quality.

The superiority of well-specified CO-STAR prompts is attributable to the structural guidance they provide to the model. When a prompt fixes the persona as a senior Solidity developer, names payments, conditions, constraints, and termination as the aspects of the agreement that the code must reflect, and specifies that the output must target Ethereum, require Solidity pragma > 0.8, include fully defined

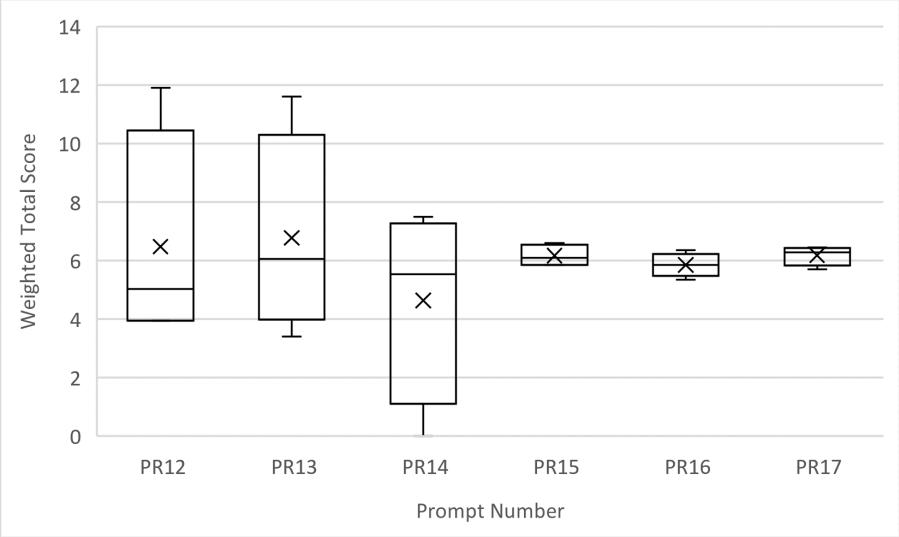


Fig. 4.2 Weighted Total Score boxplot about *4* calls to ChatGPT's API with the same prompt.

function logic, initialize all parameters from the input document, and produce compilable secure code, the model is constrained to a substantially narrower region of its generation space. Conversely, a prompt that merely presents the legal agreement text with minimal framing leaves the model to resolve ambiguities about output format, detail level, and security requirements through its own statistical priors, which introduces variance. These findings strengthen and extend the general principle established in the prompt engineering literature that structured, objective-driven prompts systematically outperform unstructured ones for code generation tasks [76, 80].

4.5.4 Discussion

The 98.1% compilability rate and the absence of a systematic temperature trend combine to support the view that GPT-4 can function as a reliable automated code generator for smart contracts when provided with appropriately structured instructions. However, in addition to these encouraging results, the study surfaces a structural limitation that motivates the remainder of the research program presented in this thesis.

The benchmarking suite assesses contract quality entirely through automated, computationally tractable metrics: compilability, Slither-detected vulnerability counts, Solidity version, number of contracts and functions, comment presence, and initial-

ized parameter count. These metrics are valuable precisely because they are fast, reproducible, and quantifiable without human intervention. However, they are silent on the question of whether the generated contract correctly implements the *business logic* of the input agreement. A contract may compile without errors, declare a recent Solidity version, and pass Slither's checks while still failing to capture an essential clause of the underlying agreement, for instance, omitting the late-fee logic, misrepresenting the deposit refund conditions, or conflating lease termination conditions. Slither's binary vulnerability verdict provides no information about such semantic failures.

This gap between automated metrics and semantic correctness is not unique to smart contract generation. It mirrors the well-known disconnect between surface-level code quality indicators and the deeper notions of correctness, fitness for purpose, and adherence to specification that expert review can detect. Addressing this gap requires a different kind of evaluation: one grounded not in automated tool outputs but in human expert assessment of whether the generated contract accurately and completely captures the contractual logic of the input document. The design and application of such an expert evaluation framework applied to 68 GPT-4-generated contracts derived from a property rental agreement and centered on the manual review of functional correctness, parameter initialization, and business logic completeness is the subject of the following chapter, which demonstrates that critical flaws such as frozen funds, missing functions, and currency mismanagement remain entirely invisible to automated tools such as Slither.

4.6 Conclusions

This chapter presented an LLM-based pipeline for the automatic generation of Solidity smart contracts from legal agreements, along with a multi-metric benchmarking suite for automated quality assessment. The pipeline operates through a single interaction with OpenAI's `gpt-4-0125-preview` model, using prompts structured according to the CO-STAR framework. Experimental evaluation on a property rental agreement demonstrated that GPT-4 can generate compilable Solidity code in 98.1% of invocations, that mean Weighted Total Score exhibits no systematic trend across temperature values in the practical range $[0.0, 1.0]$, and that structured CO-STAR prompts with explicit Objective and Requirements dimensions consistently

outperform underspecified prompts in both average quality and inter-invocation reproducibility.

These results establish that the LLM-based pipeline can serve as a practical tool for non-technical users seeking to translate legal agreements into smart contracts. At the same time, they expose the ceiling imposed by purely automated evaluation: the benchmarking suite's metrics cannot detect logical flaws in the generated contract logic, incomplete clause translation, or semantic deviations from the input document. Addressing this limitation through expert-driven evaluation frameworks constitutes the central challenge taken up in the chapters that follow, where the analysis is deepened through pattern-based manual assessment and extended to a broader multi-model, multi-agreement experimental setting.

Chapter 5

Manual Expert Evaluation of LLM-Generated Smart Contracts

5.1 Introduction

The preceding chapter established that OpenAI's GPT-4 model, guided by CO-STAR-structured prompts, is capable of generating compilable Solidity smart contracts from legal agreements at a rate exceeding 98%, and that the Weighted Total Score remains stable across a broad range of temperature settings [37]. Those results, while encouraging, were obtained exclusively through automated metrics: compilability verdicts from the Solidity compiler, vulnerability counts from Slither [19], and structural indicators such as function and comment counts. As noted in Section 4.5.4, such metrics are systematically silent on whether the generated contract correctly implements the *business logic* of the underlying legal agreement. A contract may compile successfully, pass a static analyzer, and declare a recent Solidity version while simultaneously containing logic errors that are invisible to any automated tool yet immediately consequential to any party who deploys and funds the contract.

This chapter addresses this gap directly. It presents a large-scale expert manual evaluation of 68 Solidity smart contracts generated by GPT-4 from a property rental agreement, complementing the automated analysis described in Chapter 4 with a structured review of functional correctness, parameter initialization fidelity, and business-logic completeness. The 68 contracts were obtained by submitting 17 distinct CO-STAR prompts to the `gpt-4-0125-preview` endpoint four times each

at a fixed temperature of 0.5, as described in Section 4.4.2. The two chapters share part of this material rather than an identical corpus. The shared runs are the 24 contracts produced by the four repetitions of prompts PR12 to PR17, which constitute the consistency experiment of Section 4.4.4 and account for 24 of the 54 requests reported in Chapter 4; the remaining 30 requests of that chapter sweep the temperature parameter over the same six prompts and lie outside the present corpus, while the 44 contracts generated here by prompts PR1 to PR11 were not examined in Chapter 4. The automated metrics reported in Section 5.4 were therefore recomputed over the full set of 68 contracts. This dual-evaluation design makes it possible to quantify the extent to which automated quality signals agree with or diverge from expert judgment, and to characterize the categories of flaw that automated tools systematically fail to detect.

The evaluation reveals a troubling disjunction between automated scores and actual contract quality. Many contracts that score well on the Weighted Total Score contain serious logical errors: funds irreversibly locked inside the contract with no withdrawal mechanism available to the landlord, security deposit functions absent altogether, late-fee logic omitted or mis-implemented, and all monetary values expressed in Ether rather than in the US dollars specified by the agreement. None of these errors trigger Slither warnings. The conclusion is unequivocal: the current iteration of GPT-4, even when prompted according to the best-practice CO-STAR methodology, is *incapable* of generating production-ready smart contracts for legal agreements of real-world complexity.

The remainder of this chapter is organized as follows. Section 5.2 situates the expert evaluation in the context of previous work. Section 5.3 describes the experimental methodology, including the test document, the prompt suite, and the evaluation criteria. Section 5.4 reports the results of the automated benchmarking. Sections 5.5 through 5.7 present expert evaluation of functional logic, parameter initialization, and code comments. Section 5.8 synthesizes the findings. Section 5.9 concludes and identifies directions for future research.

5.2 Related Work

The landscape of previous work on the generation and evaluation of LLM-assisted smart contracts was surveyed in Section 4.2. The present section focuses on the more

specific question of how prior work has treated the evaluation of generated contract quality and identifies the methodological gap that this chapter fills.

Automated evaluation through static analyzers is the dominant approach in related work on smart contract generation. Qasse et al. [96] reported compilability and Slither vulnerability counts as the primary indicators of contract quality in their Chat2Code system. Petrovic and Al-Azzoni [97] also relied on compilability as the main success criterion. Hao et al. [95] assessed generated contracts using a combination of code template similarity metrics and manual inspection of a small subset, but did not report a systematic review of function-level. The absence of a rigorous function-level expert evaluation is a consistent limitation of this body of work: automated metrics provide reproducible and scalable indicators of surface-level quality, but they cannot assess whether the contract accurately encodes the business logic of the source agreement.

The need for expert evaluation in code generation tasks has been recognized more broadly in the software engineering literature. Sobania et al. [80] noted that automated test-suite pass rates are insufficient to capture the full range of behaviors a repaired program might exhibit, and advocated manual inspection of model outputs as a necessary complement to automated metrics. Liu et al. [74] argued that prompt engineering studies that rely exclusively on automatic metrics risk overfitting the evaluation to the specific vulnerability patterns that tools are designed to detect, missing systematic errors of omission or semantic divergence. In the smart contract domain specifically, the gap between static analysis results and expert assessment has received relatively little systematic attention. Tonelli et al. [99] proposed a framework of software metrics for smart contracts, including several structural metrics that can be automatically computed, but noted that contract correctness with respect to a legal specification requires human judgment that metrics alone cannot provide. The present chapter addresses this gap by introducing a structured expert evaluation framework applied to a corpus of 68 GPT-4-generated contracts, providing the most comprehensive manual assessment of LLM-generated smart contract quality reported in the literature to date.

5.3 Experimental Methodology

5.3.1 Test Document and Contractual Scope

The source legal agreement is the property rental agreement introduced in Section 4.4.1, a standard lease contract for a residential property in Miami, Florida, involving a landlord (David Miller) and a tenant (Richard Garcia), with a monthly rent of $850 USD, a security deposit of $1 700 USD, a late fee of $200 USD per occurrence assessed after the tenth day past the due date, and a fixed lease term running from March 1, 2022, to March 1, 2023. This agreement was selected for its diversity of parameter types (numerical, textual, address, date) and its inclusion of non-trivial conditional logic governing deposit refund, late-fee assessment, and early termination. As noted there, these characteristics provide a comprehensive stress test of the model's ability to extract and translate heterogeneous contractual information into executable Solidity code.

From this agreement, a team of two domain experts identified five core contractual functionalities that any correct smart contract implementation must provide: rent payment forwarding to the landlord, security deposit management functioning as an escrow mechanism, lease activation and termination management, late-fee assessment integrated into or complementary to the payment logic, and currency management accounting for the USD-denominated terms of the agreement. These five functional categories form the basis for the expert evaluation framework applied in Sections 5.5 through 5.7.

5.3.2 Prompt Suite and Generation Pipeline

The 17 prompts and the generation pipeline are described in detail in Sections 4.4.2 and 4.4. Briefly, each prompt was constructed around the CO-STAR framework, varying the assumed developer role (Master's student, junior developer, senior Solidity developer), the objective formulation, the stylistic register, the intended audience, and the presence or absence of the two auxiliary clauses, namely the explicit software requirements and the incentive statement, as itemised in Table 4.2. Each of the 17 prompts was submitted to the `gpt-4-0125-preview` endpoint four

times at a fixed temperature of 0.5, producing 68 smart contracts in total. All 68 contracts are publicly available on the accompanying GitHub repository[1]

5.3.3 Evaluation Criteria

The automated evaluation applied the seven-metric Weighted Total Score (WTS) framework described in Section 4.3.4, covering compilability, Slither-detected vulnerability severity, Solidity version, number of contracts, number of functions, presence of comments and number of initialized parameters. The expert evaluation complemented these automated indicators with a structured manual inspection of each contract along three dimensions: the presence, completeness, and logical correctness of each expected function; the fidelity of parameter initialization to the values specified in the source agreement; and the quality and informativeness of inline code comments. Expert reviewers independently examined the source code of all 68 contracts and resolved disagreements by consensus. The evaluation was conducted without reference to automated WTS scores to avoid any bias in manual evaluation.

5.4 Automated Evaluation Results

Of the 68 smart contracts examined, 64 compiled successfully, yielding a compilability rate of 94.1%. This is somewhat lower than the 98.1% rate observed in the temperature and consistency experiments of Chapter 4, a discrepancy attributable to the inclusion in the present study of all 17 prompt configurations rather than only the 6 highest-performing ones selected for detailed analysis in that chapter. The vulnerability analysis conducted with Slither showed that the majority of compilable contracts exhibited only low-severity vulnerabilities, a small proportion displayed medium-severity vulnerabilities, and only two contracts contained a high-severity finding. This vulnerability distribution is encouraging: a naive interpretation would suggest that the generation pipeline produces predominantly secure code.

The WTS values for all 68 contracts clustered between 4 and 6, with occasional peaks reaching 10 and 12 predominantly observed in responses generated by prompt PR3. A worrying observation is the apparent absence of any systematic correlation

[1] https://github.com/BChain4all/pipeline

between the specificity of the prompt and the resulting WTS. Prompts that include explicit software requirements, a senior developer, and a well-specified audience do not consistently produce higher WTS values than minimal prompts. This finding challenges the expectation, grounded in the general prompt engineering literature [75], that structured prompts with richer constraints should reliably elicit higher-quality outputs. However, as the expert evaluation results reported below demonstrate, the WTS is a poor proxy for the quality dimension that matters most: the semantic correctness of the generated contract with respect to the source legal agreement.

5.5 Expert Evaluation: Functional Analysis

5.5.1 Rent Payment Logic

The `payRent` function was present in all 68 contracts, making it the most consistently generated functional element throughout the entire corpus. Two main variants were identified. The first accepts a month identifier as a parameter and verifies that the current month's rent has not yet been paid before updating the corresponding state variable; the second accepts no parameters and verifies only that the payment arrives within the `leaseStart`–`leaseEnd` timeframe. Both variants verify that the value transferred with the transaction (`msg.value`) matches the stored `rentAmount` variable. However, this check is expressed in Ether rather than in the US dollar amount specified by the rental agreement, an issue discussed further in Section 5.8.4.

A critical and pervasive flaw was identified in the overwhelming majority of `payRent` implementations: the function does not forward the received payment to the landlord's address. The logical consequence is that the rent paid by the tenant accumulates within the contract and becomes permanently locked unless the landlord invokes a separate withdrawal function. In the rare cases where such a function exists, the contract's behavior approaches correctness; in the majority of cases, no such function is present, and the funds are effectively frozen. This is a logic error of the first order: from the perspective of any tenant who pays rent through the contract, the funds are transferred, and the obligation is recorded, yet the landlord never receives them. Slither does not detect this class of error because no known vulnerability pattern corresponds to the absence of a transfer instruction in a payment

function; the contract is structurally valid. The flaw is detectable only through expert reading of the payment flow.

5.5.2 Security Deposit Management

In sharp contrast to the ubiquity of the `payRent` function, the `paySecurityDeposit` function was absent in the majority of the contracts examined. This omission is a critical failure: without a deposit payment function, the tenant has no on-chain mechanism by which to fulfill the security deposit obligation stipulated in the rental agreement. The omission is particularly striking because every single contract declares a `securityDeposit` state variable initialized to $1 700, demonstrating that the model correctly extracts the relevant value from the agreement, yet fails to generate the corresponding function.

When the `paySecurityDeposit` function was present, it was implemented consistently: declared `external` and `payable`, restricted to the tenant via an `onlyTenant` modifier, and requiring `msg.value` to equal the `securityDeposit` variable. However, several implementations forwarded the deposit immediately to the landlord rather than retaining it inside the contract as an escrow, which directly contradicts the intent and terms of the rental agreement. The refund function (`refundSecurityDeposit`), by contrast, was present in virtually all contracts; its two main variants either refund the full deposit or allow the landlord to specify deductions. The co-occurrence of a well-formed refund function with a missing or incorrect deposit collection function constitutes a logical inconsistency that automated analysis cannot identify.

5.5.3 Late Fee Assessment

The management of late fees presented the greatest degree of omission throughout the corpus. The majority of contracts did not implement any late-fee logic, despite the rental agreement explicitly specifying a penalty of $200 per occurrence for rent paid after the tenth day. In the minority of contracts that did address late fees, two implementation strategies were identified: deduction from the security deposit via a standalone function callable only by the landlord, and integration within the `payRent`

function itself, where the total due amount is increased conditionally based on the block timestamp.

The first strategy is conceptually flawed. A tenant who repeatedly pays rent late could accumulate late fees totaling more than the $1 700 security deposit, at which point the deduction mechanism fails silently. Furthermore, in none of these implementations does the collected late fee reach the landlord, because the same frozen-funds problem identified in Section 5.5.1 applies equally here. The second strategy, integrating late-fee assessment into the `payRent` function, is the most correct approach and was found in a small number of contracts; however, even those implementations suffered from the absence of a landlord transfer instruction, making the correctly computed late-fee amount equally inaccessible.

5.5.4 Lease Lifecycle Management

Two strategies were identified to manage lease activation in the corpus. In the first and most common strategy, the constructor sets `leaseActive = true` unconditionally, meaning the lease is considered active from the moment of deployment. This approach is logically inconsistent: the rental agreement specifies that the security deposit must be paid before the lease commences, yet an instantaneously active lease removes any pre-commencement window during which the tenant could fulfill this obligation without the lease being simultaneously in force. In the second strategy, a dedicated `startLease` function, callable exclusively by the landlord, sets `leaseActive = true` only after verifying that the lease start date has been reached. This approach correctly models a deployment phase during which the security deposit can be collected and was assessed by expert reviewers as the most faithful implementation of the contractual life cycle.

Lease termination was universally implemented as a landlord–callable function under the `onlyLandlord` and `isLeaseActive` modifiers. However, none of the generated contracts imposed any additional guard conditions, such as a check that the termination date has been reached or that the required notice period has elapsed. The landlord is therefore granted unilateral and unconditional power to terminate the lease at any moment during its active period, a right that is not consistent with the terms of the source agreement. This represents a non-trivial legal risk for any tenant interacting with the deployed contract.

5.6 Expert Evaluation: Parameter Initialization

Several prompts in the suite (PR4, PR5, PR9, PR10, PR14 through PR17) explicitly requested that all parameters be assigned the values specified in the rental agreement, so that the generated contract would be deployable without requiring the deployer to supply any input values. The expert evaluation revealed a strikingly weak correlation between this explicit instruction and the actual initialization behavior of the generated contracts.

The numerical parameters were initialized more reliably than the textual ones. Monetary values such as the monthly rent ($850), the security deposit ($1 700), and the late fee ($200) were consistently assigned as integer constants in the contract state, although always expressed in Ether rather than USD. The date parameters, when initialized, were expressed as Unix timestamps in seconds, and the values used for the lease start and end dates were consistently correct with respect to the UTC representation of March 1, 2022, and March 1, 2023.

In contrast, textual parameters were poorly handled. The names and Ethereum addresses of the landlord and tenant, and the physical address of the rental property, were frequently left uninitialized or set to placeholder values even in prompts that explicitly requested full parameter assignment. Prompt PR4, for instance, included the instruction to assign all available parameters, but only one of its four invocations assigned the landlord and tenant names. Prompt PR5, despite an analogous instruction, did not assign names or addresses to any of its four outputs. This inconsistency was observed across roles (Master's student, junior developer, senior Solidity developer) and was unaffected by the specificity of the software requirements listed in the prompt.

5.7 Expert Evaluation: Code Comments

Code comments were present in the overwhelming majority of contracts, contributing positively to the WTS even though the prompts did not explicitly request their inclusion. The expert review found that the comments, while technically accurate, were uniformly terse: they identified the function being defined and summarized its purpose in one or two lines, without explaining non-obvious implementation choices,

documenting edge cases, or providing guidance for integrating the contract with off-chain components.

In several contracts, comments acknowledged the existence of unresolved implementation concerns. A comment in a contract generated in response to prompt PR1 noted that the contract "represents the landlord's balance for simplification", implying that the payment logic had been intentionally simplified in a way that obscures the frozen-funds error described in Section 5.5.1. Similarly, several `paySecurityDeposit` implementations included comments acknowledging that the security requirements would need to be reviewed by a developer, without providing any guidance on what those requirements should entail.

The consistency of comment style across all 68 contracts, independent of the role or expertise level specified in the prompt, suggests that the model's documentation behavior is largely independent of the CO-STAR context dimension. This may reflect the statistical regularities of the Solidity corpora on which GPT-4 was pre-trained: if the open-source Solidity codebase on GitHub predominantly features brief, per-function comments, the model will reproduce that pattern regardless of the role it is asked to assume [26].

5.8 Discussion

5.8.1 The Failure of Automated Metrics as Proxies for Correctness

The most consequential finding of this study is the systematic divergence between automated WTS and expert assessment of contract correctness. Contracts that score well on the WTS frequently contain critical logical errors that automated tools cannot detect. The `payRent` functions that do not forward payments to the landlord are compilable, declare no state-variable reentrancy, and trigger no Slither detector; they are, in the language of automated evaluation, high-quality code. Yet, they are fundamentally broken from the perspective of any landlord who expects to receive rent. This divergence is not incidental: it reflects a structural limitation of static analysis as an evaluation paradigm. Slither's detectors correspond to known

vulnerability *patterns*; the absence of a required transfer instruction does not match any known pattern and therefore goes unreported.

This finding has important implications for the broader agenda of automated smart contract generation. Research that evaluates generated contract quality exclusively through automated metrics risks producing misleadingly optimistic conclusions about the readiness of LLM-based tools for deployment in production contexts. The results of this chapter provide empirical evidence that expert evaluation is not merely a desirable complement to automated analysis but an indispensable component of any rigorous quality assessment framework for LLM-generated smart contracts.

5.8.2 Absence of Correlation Between Prompt Specificity and Output Quality

A second major finding is the absence of a reliable correlation between the specificity or structured richness of the CO-STAR prompt and the functional correctness of the generated contract. Prompts that specify a senior Solidity developer, include explicit software requirements, and target a technically sophisticated audience do not systematically produce contracts that more faithfully implement the business logic of the rental agreement. The `paySecurityDeposit` function is absent in the majority of contracts regardless of the prompt configuration. Late-fee logic is omitted equally across minimal and richly specified prompts. The frozen-funds error in the `payRent` function persists throughout all prompt variants.

This finding suggests that the barriers to correct smart contract generation lie not in prompt formulation but in deeper limitations of the model's understanding of the blockchain execution model. The model appears to have strong statistical priors, derived from its Solidity training corpus, about what functions a lease contract should contain and how they should be structured, and these priors are relatively insensitive to the contextual framing provided by the prompt. Where those priors encode correct behavior, the model performs well regardless of prompt specificity; where they encode incorrect or incomplete behavior, prompt engineering cannot override them.

5.8.3 Non-Detectable Logic Errors and Their Implications

The frozen-funds error is the most systematically significant logic flaw identified in this study. From the perspective of the LLM, the smart contract itself appears to play the role of the landlord: payments are received and recorded, but never forwarded. This interpretation is evidenced by a comment found in one smart contract, noting that the entire contract models the landlord's balance for simplification. The consequence is that a tenant interacting with the deployed contract fulfills their rental obligations on-chain, while the real landlord receives nothing. This is not an edge case or an exotic attack vector: it is a baseline functional failure that would be immediately apparent to anyone attempting to use the contract in a real tenancy. That it passes all automated checks underscores the severity of the gap between automated quality metrics and production-readiness.

A related class of flaw concerns the absence of genuine comprehension of the semantic of legal contracts. The rental agreement grants the landlord the right to terminate the contract only under specific conditions. The generated contracts grant this right unconditionally. While modifiers such as `onlyLandlord` and `isLeaseActive` are correctly applied, no additional guard conditions are imposed, leaving the landlord free to terminate prematurely in violation of the agreement's terms. This class of error requires legal domain knowledge to detect: a reviewer unfamiliar with the agreement's provisions would not identify it as an error. Non-expert users, who constitute the primary intended beneficiaries of automated smart contract generation tools, would therefore be unable to discover it.

5.8.4 Currency Mismanagement

All 68 contracts express monetary values in Ether despite the rental agreement's exclusive use of US dollars. The monthly rent of $850 USD becomes 850 Ether, a sum worth several million dollars at typical ETH exchange rates, rendering the contract economically non-functional under any real-world deployment scenario. No contract in the corpus includes a mechanism for USD-to-ETH conversion, a stablecoin transfer, or an oracle-based exchange rate lookup, despite the fact that the correct handling of fiat-denominated payments is a well-studied problem in the DeFi literature [6]. This systematic failure reflects the model's training distribution: the majority of Solidity examples in public repositories use Ether as the unit of account,

and the model reproduces this convention regardless of the currency specification in the source agreement.

5.9 Conclusions

This chapter presented a comprehensive expert manual evaluation of 68 Solidity smart contracts generated by GPT-4 from a property rental agreement, complementing the automated analysis of Chapter 4 with a structured function-level and parameter-level review conducted by domain experts. The evaluation identified five categories of systematic failure: frozen funds in the rent payment logic, absent or misimplemented security deposit collection, pervasive omission of late-fee logic, legally inadequate lease termination conditions, and universal currency mismanagement. None of these failures are detectable by Slither or by any other automated static analysis tool, yet all of them would be immediately consequential to any party relying on the contract to govern a real tenancy.

The central conclusion of this study is unambiguous: the present generation of GPT-4, even when prompted according to the structured CO-STAR methodology, is incapable of generating production-ready smart contracts for agreements of real-world legal complexity [30]. Automated WTS, while useful as a first-pass screening indicator, is an insufficient indicator of production readiness and must be complemented by expert evaluation. This conclusion motivates two directions for future research. The first is the design of richer evaluation frameworks that combine automated metrics with structured semantic correctness checks, moving beyond syntactic and vulnerability-pattern analysis toward specification-grounded assessment. The second is the investigation of alternative generation strategies, including few-shot prompting with verified exemplar contracts, chain-of-thought prompting that guides the model through the contractual logic step by step, and fine-tuning on domain-specific datasets of high-quality, expert-validated Solidity contracts. These directions are taken up in the chapters that follow, where the evaluation framework is extended to a multi-model, multi-agreement setting and the contributions of human-in-the-loop review are examined systematically.

Chapter 6

Light and Shadows of Smart Contract Development with LLMs

6.1 Introduction

The preceding chapters established a trajectory of growing complexity in the application of Large Language Models to smart contract engineering. Chapter 3 demonstrated that LLMs could assist with vulnerability correction in existing contracts, while Chapters 4 and 5 showed that GPT-4, guided by CO-STAR prompts, could generate compilable Solidity contracts from a property rental agreement, even though with serious shortcomings in functional completeness and security. However, both of those investigations examined a single model acting on a single, relatively simple legal agreement. The question that naturally follows is whether these results generalise: do other frontier LLMs exhibit comparable capabilities, and does performance hold when the legal agreement grows in length, conditional complexity, and domain specificity?

This chapter addresses that question through a systematic, multi-model, multi-agreement evaluation. The study assesses four leading commercial LLMs, `gpt-4-turbo` by OpenAI, `claude-3.5-sonnet` by Anthropic, `mistral-large` by MistralAI, and `gemini-1.5-pro` by Google, across five legal agreements of varying complexity: a Cleaning Service Agreement, a Compensation Agreement, an Equipment Rental Agreement, a Mortgage Note Agreement, and a Property Sale Agreement. The evaluation combines an automated vulnerability analysis performed by the Slither

static analyser [19] with a structured manual inspection of smart contract design patterns, drawing on the pattern taxonomy of Bartoletti and Pompianu [39] and Khan et al. [40]. The result is the most comprehensive comparative assessment of LLM-based legal-to-smart-contract generation published to date, providing empirical evidence on both the relative strengths of current frontier models and the structural barriers that no model has yet overcome.

The chapter is organized as follows. Section 6.2 surveys related work. Section 6.3 describes the experimental pipeline, the legal agreements examined, the prompt suite, and the evaluation framework. Section 6.4 presents results on four dimensions: overall performance, agreement-type analysis, prompt analysis, and vulnerability analysis. Section 6.5 synthesizes the findings and discusses implications for automated contract generation. Section 6.6 concludes and identifies directions for future work.

6.2 Related Work

Smart contract generation from natural language specifications has been approached through several methodological traditions. Template-based systems, such as the iSyn framework of Fang et al. [90], extract structured information from legal agreements using Natural Language Processing and populate pre-designed Solidity templates. Although effective for well-defined transaction types, such as payments and transfers, template-based approaches are inherently limited in expressiveness: Fang et al. explicitly acknowledge that iSyn cannot model late fees, deposits, or price variations. Model-driven approaches, exemplified by the MDAsmartSC methodology of Jurgelaitis et al. [100] and the SecBPMN2BC framework of Kopke et al. [101], transform abstract process models into smart contract code through formal transformation rules, but require users to invest significant effort in learning the modeling notation before any contract can be produced. Visual-oriented approaches, such as the Blockly-based system of Trestioreanu et al. [59] and the Char-RNN framework of Mao et al. [102], lower the entry barrier through drag-and-drop interfaces but sacrifice expressiveness for complex contractual logic.

LLM-based generation has attracted increasing attention since the public release of ChatGPT in late 2022. Qasse et al. [96] proposed Chat2Code, a chatbot-driven system in which conversational interaction populates an iContractML instance that

is subsequently used to generate Solidity code, but this approach requires users to familiarize themselves with iContractML terminology. Petrovic and Al-Azzoni [97] used ChatGPT to generate contracts from iContractML meta-models without dedicated security verification. The preliminary evaluation described in Chapter 4 and the expert manual evaluation reported in Chapter 5 assessed GPT-4 acting on a single property rental agreement, finding compilability rates of approximately 98% along with serious shortcomings in business logic completeness. Baralla et al. [29] evaluated GitHub Copilot on standard token contracts, finding acceptable performance on simple ERC-20 and ERC-721 patterns but significant failures on complex logic and security patterns. The work reported in this chapter extends all of these studies by comparing four frontier models acting on five agreements drawn from distinct domains, using an evaluation framework that combines automated static analysis with structured manual pattern inspection.

6.3 Methodology

6.3.1 Generation Pipeline and Models

The generation pipeline follows a five-stage architecture, illustrated in Figure 6.1. A plain text legal agreement provided by the user is transformed into a structured prompt using the CO-STAR framework [38], which encodes the Context, Objective, Style, Tone, Audience, and Response dimensions. The consolidated prompt is sent to the LLM API endpoint, and the returned text is saved as a `.sol` source file. Slither [19] then performs a static vulnerability analysis on each generated contract. Finally, compilable contracts are subjected to structured manual evaluation according to the pattern framework described in Section 6.3.3.

The four LLMs evaluated are `gpt-4-turbo`, `claude-3-5-sonnet-latest`, `mistral-large-latest`, and `gemini-1.5-pro`. Six of the seventeen CO-STAR prompts defined in the Appendix A (PR12–PR17) were selected for evaluation to manage endpoint costs. Each prompt was submitted four times per model per agreement following the consistency methodology of Sobania et al. [80], yielding a total of 480 endpoint calls (6 prompts $\times$ 5 agreements $\times$ 4 LLMs $\times$ 4 API calls). All contracts were generated in September 2024; manual evaluation was conducted

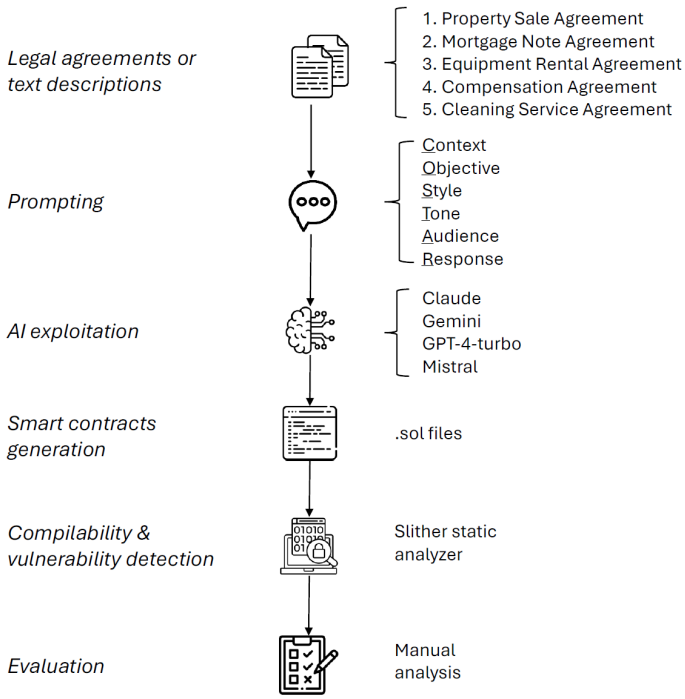


Fig. 6.1 Pipeline used to generate smart contracts from legal agreements.

between October and December 2024. The pipeline code and generated contracts are publicly available.[1]

6.3.2 Legal Agreements

Five legal agreements were sourced from the PandaDoc[2] template library, covering domains where blockchain-based automation offers concrete benefits in terms of immutable record-keeping, transparent payment trails, and verifiable contractual state. The evaluation is carried out on five legal agreements, named:

1. Property Sale Agreement[3]
2. Mortgage Note Agreement[4]
3. Equipment Rental Agreement[5]

[1] https://github.com/BChain4all/pipeline
[2] https://www.pandadoc.com/agreement-templates/
[3] https://www.pandadoc.com/kentucky-real-estate-purchase-agreement-template/
[4] https://www.pandadoc.com/mortgage-note-template/
[5] https://www.pandadoc.com/equipment-rental-agreement-template/

4. Compensation Agreement[6]

5. Cleaning Service Agreement[7]

All legal agreements can be found in markdown format in the Github repository.[8] Table 6.1 reports their word counts and estimated token counts.

Legal Agreement	Word Count	Avg. Token Count
Cleaning Service	859	1,573
Compensation	550	1,019
Equipment Rental	1,305	2,042
Mortgage Note	1,239	1,958
Property Sale	1,140	2,019

Table 6.1 Word and estimated token counts for each legal agreement (assuming 1 token $\approx$ 4 characters on average).

The agreements span a meaningful range of structural complexity. The Compensation and Cleaning Service agreements are characterized by short length, clear terminology, and linear workflow logic. The Equipment Rental and Mortgage agreements introduce multi-party dependencies, financial calculations with penalties and late fees, and time-constrained obligations. The Property Sale Agreement adds contingency clauses, inspection conditions, and escrow mechanics that are particularly difficult to translate into deterministic executable logic.

6.3.3 Evaluation Framework

A critical review of the evaluation metrics employed in Chapter 4 identified several systematic limitations [30]: the preliminary metrics did not capture missing function bodies, the absence of fund-withdrawal mechanisms, or the widespread substitution of Ether for the USD-denominated values specified in the agreements. In response, this chapter adopts an evaluation framework grounded in the taxonomy of smart contract design pattern of Bartoletti and Pompianu [39] and Khan et al. [40]. Ten patterns are assessed for each generated contract: *Token*, *Authorization*, *Oracle*, *Math*, *Randomness*, *Poll*, *Termination*, *Time Constraint*, *Fork Check Helper*, and *Helper*. An eleventh pattern, *Business Logic*, was introduced to assess the precision

[6] https://www.pandadoc.com/compensation-agreement-template/
[7] https://www.pandadoc.com/cleaning-service-contract-template/
[8] https://github.com/BChain4all/pipeline/tree/main/test_contracts_txt

and completeness of contract-specific functional requirements derived from the source legal agreement. 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;
- *Authorization* 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;
- *Poll* 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 Helper* 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;
- *Business Logic* patterns assess the accuracy and completeness of the contract's core functionality as it relates to the intended business rules and processes.

Each pattern is scored on a five-point ordinal scale, where 0 denotes that no requirements are met, and 4 denotes that all requirements are met. Manual evaluation was conducted only on compilable contracts; non-compilable contracts receive a null score for all patterns. To reflect the conservative posture appropriate for smart contract assessment, the aggregate score across four runs of the same prompt is computed as

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

where x_i is the score assigned in the run i and $\lfloor \cdot \rfloor$ denotes the floor function. This approach deliberately penalizes inconsistency: a model that produces one excellent contract and three inadequate ones is assigned a low aggregate score, reflecting the deployment risk that inconsistency implies. The evaluation was conducted by a team of two experts in the blockchain domain after calibration sessions designed to align the scoring criteria between the experts.

6.4 Results

6.4.1 Overall Performance and Model Ranking

The aggregated results, presented as stacking bar charts in Figures 6.2 and 6.3, reveal a stable performance hierarchy between legal agreements and prompt configurations: Claude > GPT-4-Turbo > Gemini > Mistral. This ordering holds for the overall pattern score and for the performance on the most demanding agreement types. On compilability it holds only in part: Claude and GPT-4-Turbo tie at 88.3% (106 of 120 contracts each, as reported in Table 6.2), and the ordering separates that leading pair from Gemini and Mistral rather than ranking the two leaders against each other.

`claude-3-5-sonnet-latest` achieves 100% compilability for the Cleaning Service and Compensation agreements across all prompts, and maintains strong scores on the Authorization, Termination, and Business Logic patterns. Its performance degrades on the Mortgage Note Agreement, where compilability drops to 25–100% depending on the prompt, and it fails to produce a compilable contract for the PR12 prompt within the Property Sale Agreement. `gpt-4-turbo` is a consistent second, achieving compilability rates of 75–100% across most configurations and performing particularly well on Termination and Business Logic

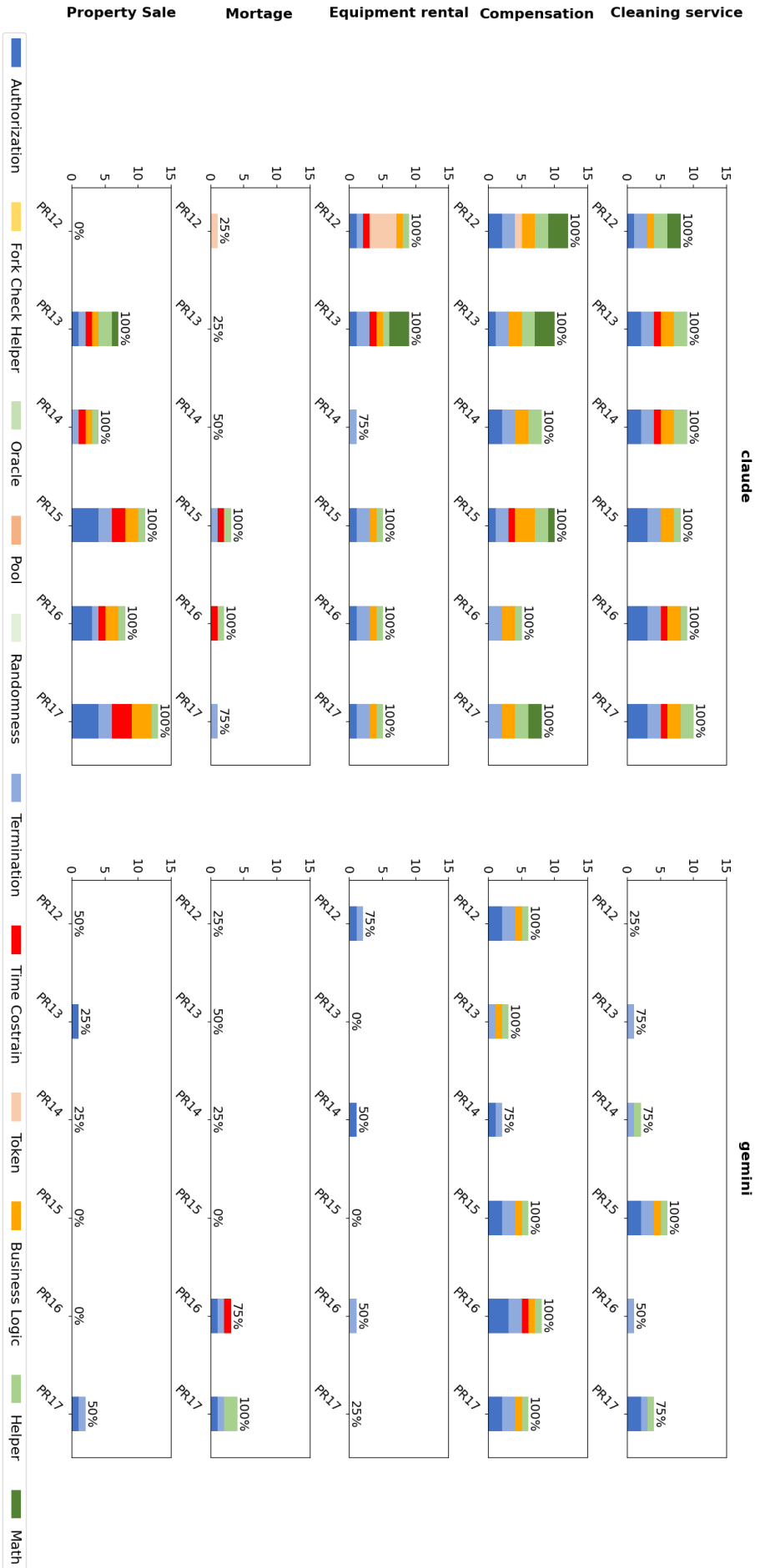


Fig. 6.2 Aggregated results obtained by averaging the score of each pattern of smart contracts generated by Claude and Gemini.

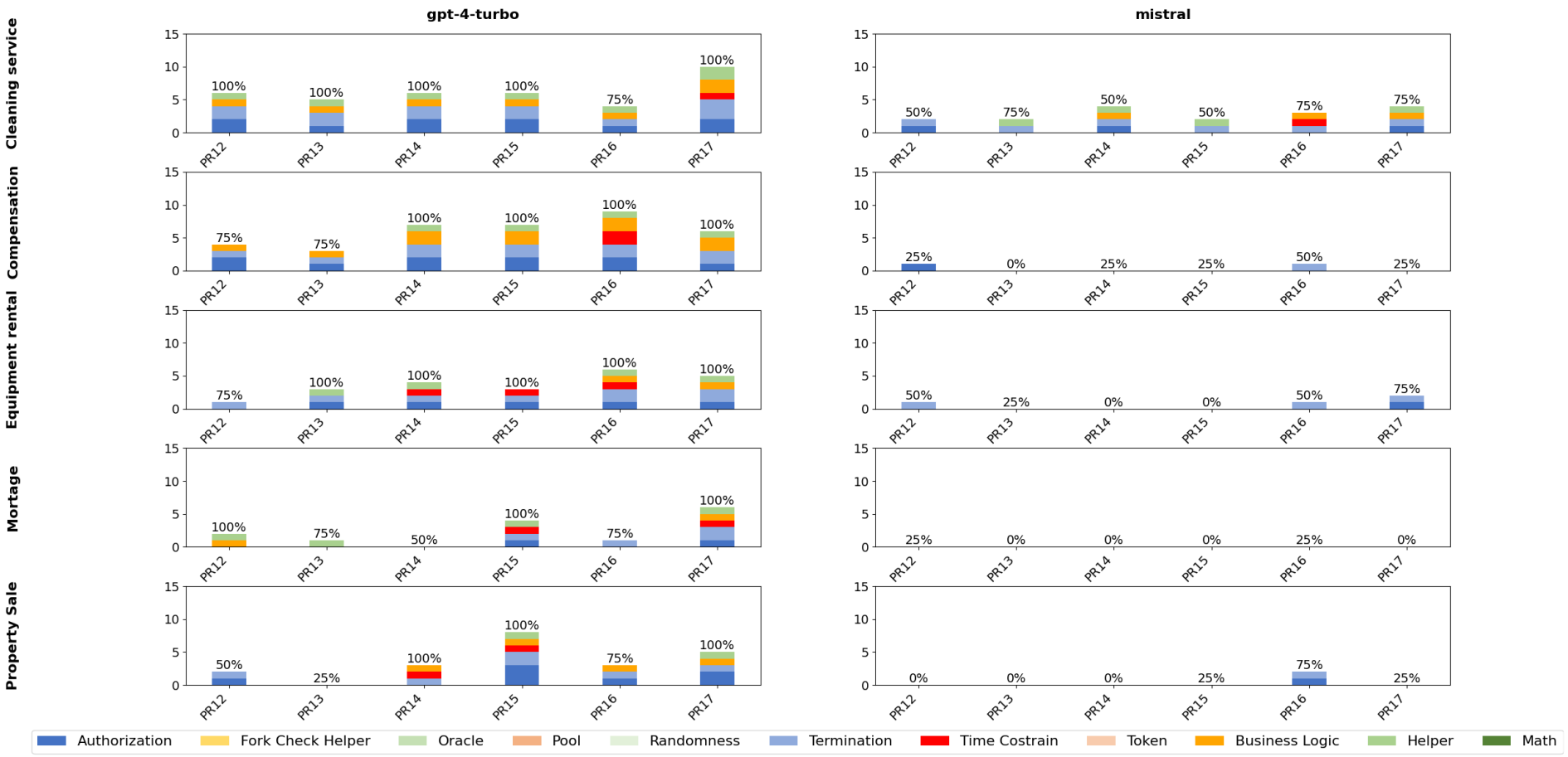


Fig. 6.3 Aggregated results obtained by averaging the score of each pattern of smart contracts generated by GPT-4-Turbo and Mistral.

patterns, though it struggles with the intricate conditional logic of the Property Sale and Mortgage agreements. `gemini-1.5-pro` achieves moderate and inconsistent results, ranging between 50–75% compilability with occasional peaks of 100% on the Compensation agreement, and shows persistent difficulty with Business Logic patterns. `mistral-large-latest` demonstrates the weakest performance, with compilability rates frequently below 50% and near-zero pattern scores even for agreements where other models perform well.

6.4.2 Agreement Complexity and Non-linear Performance Degradation

A key finding of this study is that the relationship between legal agreement complexity and model performance is non-linear. Performance degradation does not scale proportionally with agreement length [1]. The Cleaning Service and Compensation agreements are handled adequately even by lower-performing models, while the Mortgage and Property Sale agreements cause significant failures even in top-performing ones. For example, Claude achieves 100% compilability across all prompts for the Compensation Agreement but only 25–50% for the Mortgage Agreement on several prompts, despite the Mortgage Agreement being only modestly longer in terms of token count (Table 6.1).

This non-linearity can be attributed to the inherent ambiguity of natural language legal texts, which encompasses lexical ambiguity (words carrying multiple meanings), syntactic ambiguity (phrases admitting multiple grammatical readings), antecedent ambiguity (unclear pronoun references), temporal ambiguity (vague time indications), and contract reference ambiguity (terms such as “hereunder” or “reasonable”) [103]. These ambiguities are not incidental: they are often deliberately introduced by legal writers to preserve interpretive flexibility [104]. For a language model, which generates output by predicting statistically likely continuations of input sequences, each encounter with an ambiguous clause constitutes a potential divergence point. Balaji et al. [105] demonstrated that LLMs generate their own interpretations of ambiguous legal clauses when instructed to convert legal agreements to smart contracts, producing different interpretations across runs. This explains the variability observed within the model in the present study, particularly for the Mortgage and Property Sale agreements.

The temperature parameter was not identified as a contributing factor. Indeed, previous work [106, 107] reports a negligible effect of temperature on code generation quality, and the exploratory analysis in Chapter 4 observed no systematic trend in mean WTS across the range $[0.0, 1.0]$. The non-linear degradation observed here is therefore attributable primarily to the structural properties of the legal documents rather than to model stochasticity.

6.4.3 Prompt Design and Its Effect on Generation Quality

Analysis of the six evaluated prompts (PR12–PR17) reported in Figure 6.5 reveals a generally increasing trend in the aggregate pattern score from PR12 to PR17, with the most substantial gains appearing in the Authorization, Business Logic, and Time Constraint patterns. Prompt PR17, which specifies an expert audience (the company's legal department), explicit software requirements including compilability, security, and full function definitions, and a formal professional style, achieves the highest aggregate score across models and agreement types. The improvement between PR15 and PR17 is most pronounced in Claude and GPT-4-Turbo; Gemini and Mistral show more modest and less consistent gains.

An important qualification is that prompts designed for a more demanding audience do not uniformly improve results across all models and agreement types. For the Mortgage Agreement, performance remains relatively low between all prompts and all models, as can be seen in Figures 6.2 and 6.3, suggesting that some document types may require specialized prompt structures that the general CO-STAR framework cannot provide. Prompts PR15 and PR16 generate compilable contracts in 75–100% of cases for most model–agreement combinations, with the exception of Mistral and three agreement types for Gemini.

This study adopts a one-shot prompting methodology in order to simulate the conditions faced by a non-expert user with no background in either legal drafting or smart contract programming. Although this choice is appropriate for the stated research objective, it imposes a ceiling on achievable quality. The potential of chain-of-thought (CoT) that prompts substantial improvement results is investigated in Section 6.4.5.

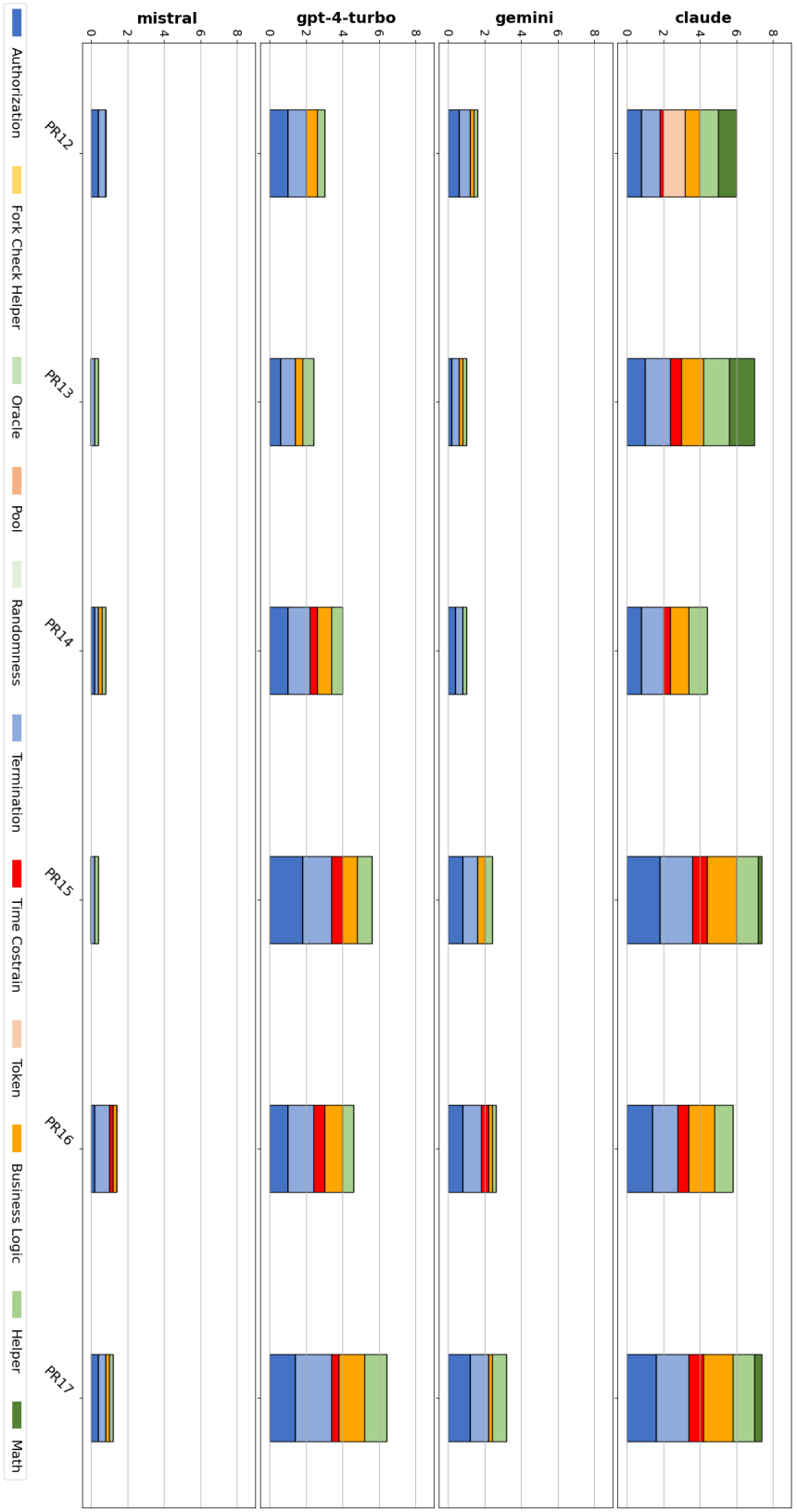


Fig. 6.4 Aggregated pattern results for each LLM used

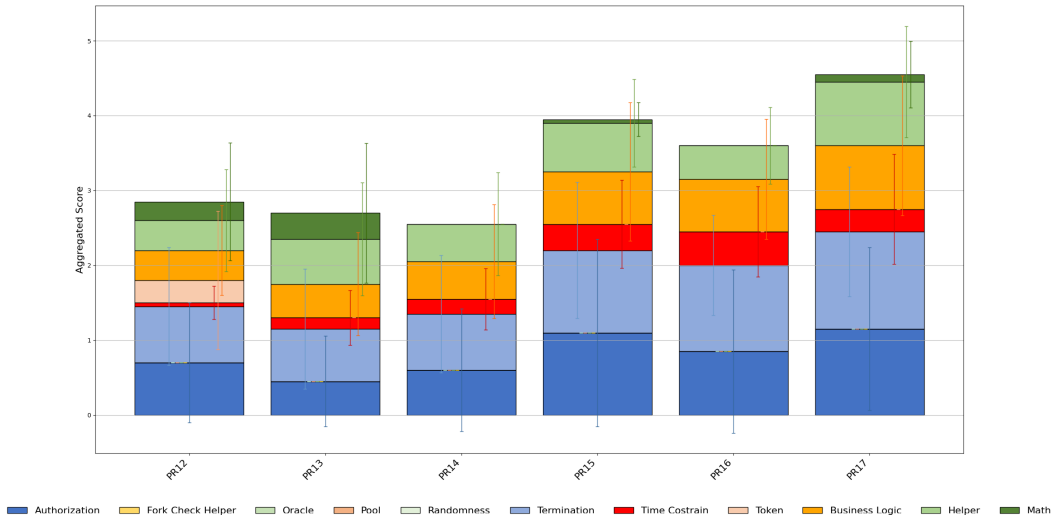


Fig. 6.5 Variation of pattern score for each prompt

6.4.4 Vulnerability Analysis

Table 6.2 summarizes the vulnerability assessment performed by Slither between all generated contracts.

LLM	High	Medium	Low	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	35/120 (29.2%)

Table 6.2 Vulnerability counts identified by Slither, aggregated by LLM, with compilability rates.

Claude and GPT-4-Turbo share an identical compilability rate of 88.3% (106 of 120 contracts), and both exhibit substantially higher vulnerability counts than Gemini and Mistral, as can be observed in Figure 6.6. This correlation is not paradoxical once the causal mechanism is understood: Slither can only analyze compilable contracts, and a model that produces more compilable contracts exposes more of its output to static analysis, producing higher raw vulnerability counts. Normalizing by the number of analyzable contracts bears this out for Mistral, which yields 3.46 findings per compilable contract against 3.85 for Claude and 3.98 for GPT-4-Turbo: its low total reflects reduced exposure to analysis rather than safer code. Gemini, however, records 1.69 findings per compilable contract, under half the rate of the two leading models, so exposure alone does not account for its lower count. The present

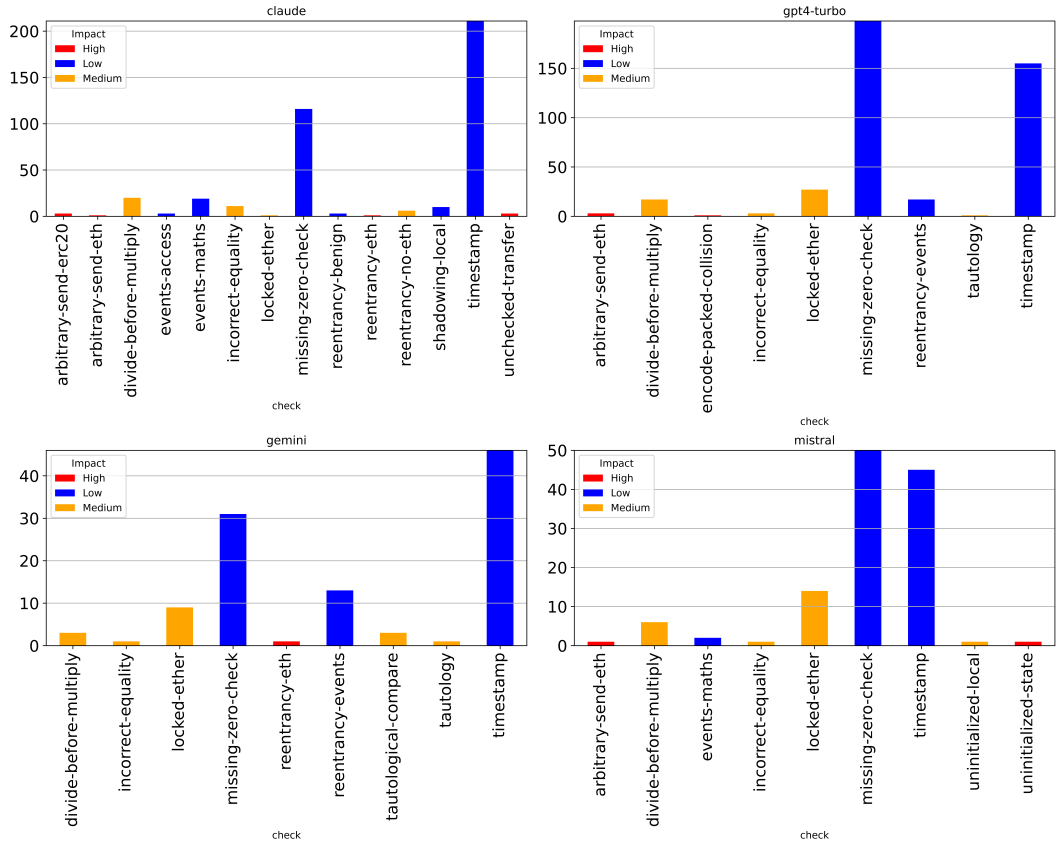


Fig. 6.6 Type of vulnerability detected by Slither in the generated smart contracts.

data cannot settle whether this residual gap reflects a more conservative generation style or simply the narrower range of Solidity constructs Gemini produces; in neither case should the lower raw counts of the two weaker models be read as evidence of superior security practices.

Six distinct high-impact vulnerability classes were detected across the four models taken together, although no single model exhibits all six: Claude accounts for four of them, GPT-4-Turbo and Mistral for two each, and Gemini for one. The classes are the following:

- *arbitrary-send-erc20* arises from insufficient validation of the `from` address in `transferFrom` calls;
- *arbitrary-send-eth* permits unauthorized Ether withdrawals through unprotected transfer functions;

- *unchecked-transfer* results from failure to inspect the return value of `transfer`/`transferFrom` calls, enabling silent token loss;
- *encode-packed-collision* occurs when `keccak256` is applied to `abi.encodePacked` output containing multiple dynamic types;
- *uninitialized-state* arises from state variables not being explicitly initialized;
- *reentrancy-eth* allows reentrant calls during Ether transfers, enabling state manipulation and fund drainage.

These findings confirm that LLM-generated contracts inherit the security patterns of the training corpus: models trained on the publicly available Solidity code reproduce the vulnerability patterns prevalent in that corpus [19].

6.4.5 Chain-of-Thought Prompting: A Preliminary Investigation

To investigate whether more structured prompting techniques can mitigate the limitations of the one-shot approach, an exploratory experiment was conducted using Chain-of-Thought (CoT) prompting on the most challenging model–agreement combination: `mistral-large-latest` generating a smart contract from the Mortgage Note Agreement using prompt PR14, which had yielded near-zero pattern scores in the one-shot evaluation [1].

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:

- **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[a] >

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

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

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

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

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

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 CoT prompt provides the model with a role assignment as a senior Solidity developer, a worked example consisting of a rental agreement paired with its corresponding smart contract, and the target task of generating a mortgage smart contract from the actual agreement. Detailed functional requirements are specified as intermediate reasoning steps. The resulting contract, available in the companion GitHub repository [9], is compilable and implements a substantially larger number of required patterns than any one-shot Mistral contract for the same agreement. Specifically, the Token pattern is realized through USDT integration via the ERC-20 interface; the Authorization pattern is implemented through role-specific modifiers (*onlyMortgagor*, *onlyMortgagee*, *validUSDTAllowance*); the Time Constraint and Helper patterns are correctly instantiated; and the Termination pattern is partially implemented. Patterns absent from all one-shot contracts, including Token and Time Constraint, are present in the CoT-generated contract, representing a categorical improvement.

CoT prompting has demonstrated effectiveness in software engineering tasks in a more general way [108–110], and the present result suggests that this holds for smart contract generation even in the absence of a large corpus of legal-to-Solidity examples.

[9]https://github.com/BChain4all/pipeline/tree/main/Chain-of-Thought-Results

6.5 Discussion

6.5.1 Structural Limits of the LLM-Based Approach

The results of this chapter, taken together with the findings of Chapters 4 and 5, allow several structural conclusions to be drawn about the current state of smart contract generation based on LLMs.

First, achieving syntactic correctness, that is, producing compilable Solidity code, has become feasible for state-of-the-art models acting on simpler agreements. Claude and GPT-4-Turbo achieve compilability rates approaching 90% across the entire dataset. However, compilability is a necessary but not sufficient condition for contract quality. Manual evaluation consistently reveals that compilable contracts can be functionally incomplete, logically incorrect, and financially hazardous to deploy.

Second, translating complex legal semantics into executable contract logic remains a fundamental unsolved problem. The non-linear degradation observed across agreement types is not simply a scaling issue that can be resolved by using longer context windows or more powerful models. It reflects the fact that legal language is inherently ambiguous and that deterministic smart contract execution is inherently unforgiving of ambiguity. A legal clause that is deliberately vague to allow room for negotiation cannot be faithfully translated into a deterministic function that runs without human intervention.

Third, the prevalence of high-impact security vulnerabilities across all generated contracts, including those from the highest-performing models, indicates that the training data bias problem is substantial. LLMs trained on publicly available Solidity code inherit the security patterns of that code, which contain a documented high proportion of vulnerable contracts [17]. Without specialized fine-tuning on audited, vulnerability-free code, this bias is unlikely to be corrected by general capability improvements.

6.5.2 Automation Bias and the Necessity of Human Oversight

A specific risk that the results of this chapter foreground is automation bias [111]: the tendency of users to uncritically accept the output of automated systems, particularly

when that output appears technically competent. A smart contract that compiles successfully, passes a Slither analysis without high-severity warnings, and implements recognizable Solidity patterns is likely to be accepted as correct by a non-expert user, even if it locks all deposited funds permanently, omits the late-fee logic specified in the agreement, or handles all monetary values in Ether rather than the agreed currency. The more impressive a model's performance on simple contracts, the greater the risk that users will extend unwarranted trust to its handling of complex agreements.

Choi et al. [112] have shown that even domain experts can be anchored to and biased by LLM output when reviewing generated content. Effective mitigation therefore requires more than the provision of expert review: it requires explicit uncertainty indicators, mandatory human review protocols, and structured education about the specific failure modes of LLM-to-Solidity translation. A promising architectural direction, discussed by Napoli et al. [1], is the combination of LLM generation with template-based verification: the LLM populates function-level logic within a pre-audited structural skeleton, limiting its potential for introducing both logical errors and security vulnerabilities.

6.6 Conclusions

This chapter presented a comprehensive multi-model, multi-agreement evaluation of LLM-based smart contract generation from legal texts. The study evaluated four frontier commercial models in five legal agreements using a structured pattern-based evaluation framework combining automated static analysis and expert manual inspection.

The main findings are as follows. A clear performance hierarchy exists among current frontier models, with Claude and GPT-4-Turbo consistently outperforming Gemini and Mistral on all metrics. Performance degrades non-linearly with agreement complexity, with even the best-performing models showing substantial failures on Mortgage and Property Sale agreements. High-impact security vulnerabilities are present in the output of all evaluated models, including the best-performing ones, and their prevalence is correlated with compilability rather than with any intrinsic property of model quality. Prompt engineering can improve results meaningfully, with later-generation prompts targeting expert audiences that yield higher pattern

scores, but no prompt design tested was sufficient to overcome the fundamental difficulty of complex legal-to-code translation. CoT prompting shows considerable promise and produces qualitatively stronger contracts even from the weakest model on the most difficult agreement.

Together with the findings of Chapters 4 and 5, these results establish that current LLMs are not yet suitable for unsupervised smart contract generation from legal agreements of real-world complexity. They can serve as useful tools within a supervised workflow that includes mandatory expert review, structured security auditing, and careful prompt design. The path toward reliable automated generation likely requires a combination of specialized fine-tuning on verified contract corpora, hybrid template-LLM architectures, and multi-step generation pipelines that include automated validation with model refinement.

Chapter 7

Smart Contract Visualization: Solutions and Challenges

7.1 Introduction

The preceding chapters examined the extent to which Large Language Models can assist practitioners in generating and correcting smart contract source code. A complementary and equally pressing challenge concerns the *comprehension* of that code once it exists: how can the logic, state, and interactions of a smart contract be rendered intelligible to people who are not fluent in Solidity? This chapter addresses this question through a systematic mapping study of the visualization literature, charting the full landscape of formalisms, tools, and methodologies that researchers have proposed to make smart contracts easier to understand, develop, and reason about.

The motivation for this investigation is well-grounded in the literature. As established in Sections 2.2.5 and 2.2.4, the structural barriers to smart contract development are not purely linguistic: even developers who can write syntactically correct Solidity frequently lack the mental models needed to anticipate how their contracts will behave at runtime, who holds which permissions, and how state transitions relate to the obligations specified in the underlying legal agreement. Visual representations of software systems have a long history of addressing precisely this kind of cognitive gap. Block-based environments such as Scratch and Blockly have been shown to reduce cognitive load in programming education [113], while

standard notations such as the Unified Modelling Language (UML) and Business Process Model and Notation (BPMN) are deployed routinely in professional software engineering to communicate system architecture and business process logic across disciplinary boundaries.

However, the application of these established formalisms to smart contracts is not straightforward. Smart contracts possess properties that distinguish them sharply from conventional software: their immutability, their on-chain state, the economic significance of their execution, and the need to model interactions between mutually distrusting parties without a trusted intermediary. These properties motivate adaptations of existing notations and, in some cases, the design of entirely new visual languages. Understanding what has been attempted, what has proven effective and where significant gaps remain constitutes a prerequisite for any principled effort to build better tools. The present chapter provides that understanding.

This study makes the following contributions. It presents a systematic mapping of 40 peer-reviewed works published between 2018 and 2025 that propose visual representations of smart contracts. It classifies these works according to visual formalism employed, visualization category (structure, behavior, or evolution), accessibility requirements, use case domain, and the presence of user experience evaluation. It identifies temporal trends in the adoption of different approaches and characterizes the gaps that future research must address.

The chapter is organized as follows. Section 7.2 provides the conceptual background on software visualization and smart contract programming languages. Section 7.3 describes the systematic search protocol and the inclusion criteria. Section 7.4 surveys the three main families of visualization approaches identified in the corpus. Section 7.5 presents the quantitative and qualitative analysis of the findings. Section 7.6 discusses the implications of these findings and the open problems they expose. Section 7.7 concludes and identifies directions for future research.

7.2 Background

7.2.1 Software Visualization

Software visualization encompasses techniques that render programs, their execution traces, or their evolutionary histories in graphical form. Three canonical categories are recognized in the literature: structure, evolution, and behavior [114, 115]. *Structure visualization* analyzes the static composition of a software artifact, covering elements such as source code organization [116], library dependency graphs [117], and control flow representations [118]. *Evolution visualization* tracks the history of changes to illustrate how a system develops over time [119]. *Behavior visualization* draws on runtime data, such as function call sequences [79], to support performance analysis [120] and anomaly detection [121], frequently representing dynamic data as time series [122].

Block-based visual programming environments occupy a distinct position within the structure visualization tradition. By replacing textual syntax with draggable interlocking blocks, they abstract away the mechanics of a specific programming language and allow users to focus on algorithmic logic. Platforms such as Scratch and Blockly have demonstrated measurable benefits in learning text-based programming and developing computational thinking [113]. Research on cognitive load confirms that block-based interfaces reduce the information that novice programmers must process simultaneously, in part by eliminating the syntax-error channel entirely [123]. Studies involving eye tracking and galvanic skin response measurements found a lower cognitive load for university students performing simple tasks in visual environments compared to text-based languages such as Python, while performance on more complex tasks was maintained [124].

7.2.2 Smart Contract Programming Languages and Visualization Targets

The smart contract ecosystem encompasses a heterogeneous set of programming languages, each tied to a particular blockchain platform and execution model. The Ethereum platform, described in Section 2.2.2, is based primarily on Solidity, a statically typed object-oriented language whose syntax resembles JavaScript and whose

execution semantics are governed by the Ethereum Virtual Machine. Alongside Solidity, the ecosystem includes Vyper (a Python-influenced language for Ethereum emphasizing auditability), Rust (deployed on Solana and NEAR via stateless account-based models), Move (embedded in Aptos, featuring linear types that prevent asset duplication), and several other domain-specific languages such as Aiken for Cardano and SmartPy for Tezos.

Despite this diversity, Solidity remains the main high-level contract language for the blockchains that support the Ethereum Virtual Machine [125], and the literature surveyed in this chapter reflects that concentration: with few exceptions, the approaches reviewed in Section 7.4 adopt Solidity as their target representation. This is partly attributable to the size and historical primacy of the Ethereum ecosystem and partly to the relative maturity of Solidity's tooling, which provides static analysis interfaces with which visualization frameworks can integrate. The analysis carried out in this chapter reflects this distribution: almost all of the reviewed works target Solidity-based contracts, either by generating Solidity from a visual model or by visualizing the structure and behavior of existing Solidity code.

7.3 Research Methodology

The study adopts a systematic mapping approach. An initial search was performed on the SCOPUS database, which aggregates major scientific repositories including the ACM Digital Library, IEEE Xplore, Elsevier, Wiley, and Springer. The primary search string combined the term `smart contract` with terms related to visual representation: `(‘‘smart contract’’) AND ((‘‘visualization’’ OR ‘‘representation’’) OR (‘‘visual’’ AND ‘‘abstraction’’) OR (‘‘visual’’ AND ‘‘representation’’) OR (‘‘visual’’ AND ‘‘programming’’))`. This query returned 421 candidate papers, which were screened first by title and then by abstract against the following inclusion criteria.

A paper was included if it was (i) written in English, (ii) available in full text, and (iii) offered a clear visual representation of smart contracts in at least one of the following respects: the static structure of the source code, the business interactions between contracting parties, or the evolution of the contract's transition states. Papers that employed graph neural network techniques or merely presented

empirical findings in graphical form, without offering a visual programming or modeling tool for smart contracts, were excluded.

A supplementary search using the string ‘‘smart contract’’ AND (‘‘no-code’’ OR ‘‘low-code’’ OR ‘‘bpmn’’ OR ‘‘mda’’ OR ‘‘uml’’) retrieved 119 additional papers, and a secondary snowball pass through the citations of selected works was conducted. After eliminating duplicate data, 40 papers were retained for analysis. Table 7.1 summarizes all selected works with their year, visual approach, visualization category, online availability, use case, accessibility requirement level, and whether the work reported formal UX testing.

7.4 Smart Contract Visualization Approaches

The 40 selected papers can be grouped into three broad families according to the primary visualization mechanism used: block-based visual programming, model-driven approaches (predominantly BPMN and UML), and custom or domain-specific visualizations. The following subsections survey each family, with particular attention to design choices, representational goals, and degrees of public availability of the proposed tools.

7.4.1 Block-Based Visual Programming

Block-based environments adapt the paradigm pioneered by Scratch and Blockly to the smart contract domain, replacing Solidity syntax with draggable block primitives that assemble into executable contract logic. Weingärtner et al. [127] introduced an early modular Blockly framework in which contract elements are classified into types, delivery options, and contractual options, targeting industrial machinery agreements. Guida and Daniel [62] developed SolidityEditor with more than seventy custom blocks mapping to Solidity constructs, making the environment one of the most expressive Blockly-based tools evaluated. Mao et al. [102] took a generative approach, combining Blockly with neural code generation to produce function templates encapsulated in visual blocks, demonstrating early integration of machine learning with block-based tools . Merlec et al. [138] presented SmartBuilder, which organizes blocks into functional domains and augments the interface with natural language summaries intended to reduce the learning burden for non-technical

No.	Year	Reference	Visual Approach	Category	Available	Use Case (abridged)	Accessibility	UX
1	2018	[126]	JSON/HTML/REST/Pharo	Behavior	Yes	Pool contract	Basic	No
2	2018	[127]	Blockly	Structure	No	Industrial machinery contracts	Basic	No
3	2018	[128]	UML	Structure	No	Voting system	Basic	No
4	2018	[129]	UML	Evolution	Yes	Door control mechanism	Intermediate	No
5	2019	[62]	Blockly	Structure	Yes	Train insurance	Intermediate	No
6	2019	[102]	Blockly	Structure	No	Supply chain	Basic	No
7	2019	[130]	BPMN	Behavior	Yes	Invoicing, supply chain	Intermediate	No
8	2020	[131]	UML (graph grammars)	Structure	No	Escrow	Advanced	No
9	2020	[132]	Action graph	Structure	Yes	Voting system	Basic	No
10	2020	[133]	UML	Behavior	Yes	Real estate mortgage	Basic	No
11	2021	[134]	UML	Structure	Yes	Solidity library “Owner”	Advanced	No
12	2021	[135]	UML	Structure	No	Paper sheet purchase	Basic	6 users
13	2021	[136]	BPMN	Behavior	Yes	Mortgage case	Basic	No
14	2021	[137]	EFSM	Evolution	No	Airline insurance, auction	Intermediate	No
15	2021	[138]	Blockly	Structure	Yes	Payment system	Basic	No
16	2021	[61]	BPMN/YAWL	Behavior	No	Construction payment	Basic	No
17	2022	[139]	Answer Set Programming	Structure	No	Real estate	Advanced	No
18	2022	[140]	Flow-based programming	Evolution	No	6 contract scenarios	Basic	58 students
19	2022	[141]	UML	Structure	No	Voting system	Intermediate	No
20	2022	[142]	DSL	Behavior	No	Supply chain	Basic	No
21	2022	[143]	Node-RED	Behavior	No	Vaccine temperature tracking	Intermediate	No
22	2023	[144]	DCR graphs	Structure	Yes	Casino example	Intermediate	No
23	2023	[59]	Blockly	Structure	Yes	Carbon offset hook	Basic	No
24	2023	[145]	BPMN	Behavior	No	Supply chain, trade	Advanced	No
25	2023	[146]	BPMN	Behavior	Yes	X-ray exam process	Advanced	No
26	2023	[147]	BPMN + Blockly	Behavior	Yes	Crowdfunding, supply chain	Basic	No
27	2023	[101]	BPMN	Behavior	Yes	Hospital, municipality	Advanced	42 users
28	2023	[100]	UML	Behavior	Yes	Certificate issuance	Intermediate	No
29	2024	[148]	Timed automata	Behavior	No	PhD regulations, business	Advanced	No
30	2024	[149]	BPMN	Evolution	Yes	Office building construction	Intermediate	9 users
31	2024	[150]	Control flow graph	Behavior	No	Ponzi scheme detection	Advanced	12 users
32	2024	[151]	Blockly	Structure	No	NFT metadata upload	Basic	17 students
33	2024	[152]	Blockly	Structure	Yes	Smart contract coding education	Basic	N/A
34	2024	[153]	BPMN	Behavior	No	Bank loan assessment	Intermediate	No
35	2024	[154]	BPMN	Behavior	No	Certificate management	Intermediate	No
36	2024	[155]	BPMN	Behavior	Yes	Pizza order, hotel booking, etc.	Advanced	No
37	2024	[156]	BPMN	Behavior	No	Exam management	Intermediate	No
38	2025	[157]	ADOxx/SmartCML	Behavior	Yes	Decentralised auctions	Intermediate	No
39	2025	[158]	BPMN	Behavior	Yes	Land leasing	Advanced	7 students
40	2025	[159]	BPMN + LLM	Behavior	No	Voting, lending, sales, auctions	Advanced	No

Table 7.1 Selected papers in chronological order, with visual approach, category, availability, use case, accessibility requirement, and UX evaluation detail.

users. Trestioreanu et al. [59] targeted the XRP Ledger rather than Ethereum, delivering an end-to-end Blockly2Hooks pipeline from visual programming to on-chain deployment. Gomez et al. [152] introduced SmaCly with pre-defined ERC-standard templates and semantic compatibility checks, while Tsai et al. [151] designed an educational Blockly extension specifically for K-12 students learning blockchain concepts through NFT metadata upload scenarios.

A parallel line of UML-based approaches targets the structural and behavioral modeling of smart contracts rather than their direct construction. Marchesi et al. [128] extended UML diagrams with specialized stereotypes for Solidity constructs and enriched sequence diagrams with blockchain-specific annotations. Garamvolgyi et al. [129] used UML state charts to model cyber-physical systems in which physical devices interact with their on-chain representations, capturing both the evolution of device states and the associated contractual logic. Pierro [134] developed Smart-Graph, an augmented class diagram generator that includes specific Solidity constructs absent from the standard UML specification. Heckel et al. [141] constructed a visualization framework based on UML-inspired class diagrams enriched with domain-specific stereotypes for DAML templates. Jurgelaitis et al. [100] proposed MDAsmartSC, a complete model-driven architecture lifecycle framework built on UML extensions for certificate issuance scenarios. Ghaffari Saadat [131] contributed a unique symbolic approach using attributed graph grammars to represent smart contracts through typed graphs with color-coded transformation rules.

Across both block-based and UML traditions, a consistent design objective is to abstract programming complexity while preserving the semantic richness required for meaningful contract logic. Block-based systems prioritize executable code generation and practical construction, whereas UML-based frameworks emphasize architectural communication and behavioral documentation.

7.4.2 Model-Driven visualization with BPMN

BPMN has consolidated its position as the dominant visual formalism for model-driven smart contract development. Its appeal lies in a combination of factors: standardization by the Object Management Group, widespread familiarity in business process management communities, native support for multi-party collaboration

through pool and lane constructs, and the availability of mature tooling in both open-source and commercial ecosystems.

Several research groups have used BPMN choreography diagrams specifically to model the distributed, peer-to-peer nature of blockchain interactions. López-Pintado et al. [130] introduced Caterpillar, which uses single-pool BPMN models in which stakeholders appear as lanes, replacing message-based coordination with sequence flows to represent blockchain as a unified coordination mechanism. Shen et al. [155] deployed BPMN choreography diagrams with participant bands that distinguish message initiators from recipients, targeting multi-party collaborative processes. Samanipour et al. [158] selected choreography diagrams for their MDAPW3 framework precisely because they represent peer-to-peer interactions without implying a central orchestrator.

Security and adaptability have motivated the development of specialized BPMN extensions. Kopke et al. [101] augmented BPMN with blockchain-specific annotations, including symbols for enforceable obligations and execution markers on-chain, and validated the resulting notation with 42 participants in four real-world scenarios. Corradini et al. [146] developed FlexChain, a BPMN choreography framework designed to navigate the tension between the immutability of deployed contracts and the need for business process flexibility.

The transformation from BPMN models to deployable Solidity code constitutes a major sub-theme. El Abidi et al. [154, 156] developed systematic translation pipelines from BPMN elements to Solidity constructs via ranking processes and meta-model integration. Jin et al. [153] introduced a two-step transformation that passes through Prolog for formal validation before generating smart contracts in Go-language. Gao et al. [159] presented BPMN-LLM, coupling BPMN diagrams with a large language model to perform the translation, where the workflow lanes denote parties and gateway types encode conditional logic. An integration with block-based tooling was proposed by Shen et al. [147], who combined customized BPMN modeling with Google Blockly's visual programming interface, using specialized the workflow lane to distinguish participant actions from smart contract operations.

Alternative model-driven approaches include Meng et al.'s EFSMSolid framework [137], which represents contractual phases and transitions through Extended Finite State Machines, and the YAWL-based system of Ye and König [61], which exploits YAWL's formal semantics while remaining interpretable to construction-

domain professionals. Ye et al. [149] subsequently developed a three-tier visual framework that maps BPMN tasks to smart contract functions and annotates them with color-coded execution state indicators. Industry-oriented contributions include EDALoCo [143], which extends Node-RED with blockchain-oriented node types, and the work of Bodorik et al. [145], which transforms BPMN models through an intermediate event-hierarchical state machine representation to identify patterns for side chain deployment.

7.4.3 Custom and Domain-Specific visualization Approaches

A third group of works develops custom visual languages or novel interaction paradigms specifically designed for smart contract comprehension, security analysis, or domain expert communication, without anchoring to a standard notation.

Bragagnolo et al. [126] created SmartInspect, which uses a mirror-based reflection mechanism to visualize the state of a live contract through hierarchical tree structures, allowing inspection without contract redeployment. Tan et al. [132] introduced LATTE, an action-graph interface with semantic parsing capabilities that represents contract elements as interconnected nodes. Hdhili et al. [140] employed flow-based programming to render smart contracts as directed graphs of modular processing boxes, reporting improved comprehension among non-expert participants in a study involving 58 students. Bistarelli et al. [142] constructed a domain-specific language for supply chain contracts in which assets, operations, containers, and roles are mapped to standardized graphical components. Curty and Fill [157] proposed SmartCML, which uses decorated circles to represent actions and encodes read/write relationships to the contract state through explicit directional links.

For formal modeling audiences, Purnell and Schwitter [139] developed a Smart Document Editor with Answer Set Programming semantics, using color-coded bars and distinctive icons to denote logical constructs. Eshghie et al. [144] used Dynamic Condition Response (DCR) graphs with stylistically differentiated arrows to represent event relationships and constraints. Qin et al. [148] used networks of timed automata to capture temporal obligations and deadline constraints in contracts drawn from various domains, including academic regulations and technology licensing.

Several works combine multiple visual techniques within a single tool. Skotnica et al. [133] proposed a meta-model that decomposes contract logic into interrelated

diagrams that include a contract action model executed through Google Blockly; their subsequent DasContract system [136] integrates data models, process models, and form models to generate deployable Solidity code. Wen et al. [150] developed PonziLens, a hierarchical visualization system that transforms Solidity bytecode into semantic action sequences represented through color-coded charts, parallel sequence views, and node-link diagrams, specifically designed for Ponzi scheme detection in a user study involving 12 analysts.

7.5 Analysis and Findings

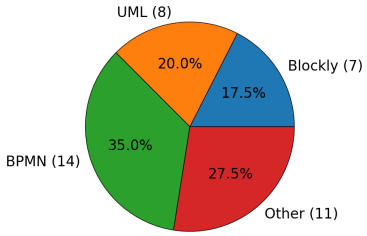
The analysis of the selected papers summarized in Table 7.1 is reported below.

7.5.1 Distribution of Visual Approaches

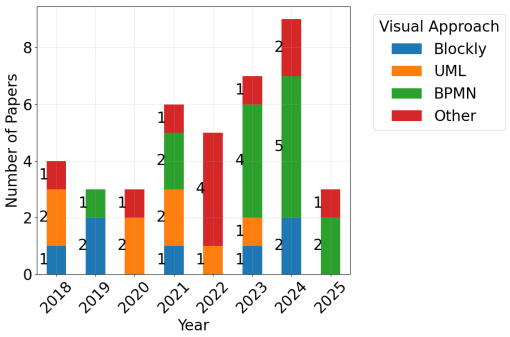
BPMN represents 35% of the selected works (14 papers) as can be seen in Figure 7.1a, making it by a clear margin the most frequently adopted visual formalism. UML follows with 20% (8 papers), while Blockly-based approaches represent 17.5% (7 papers). The remaining 27.5% is distributed across custom notations including DCR graphs, Answer Set Programming, timed automata networks, EFSM models, domain-specific languages, Node-RED, control flow graphs and ADOxx. The dominance of BPMN in the most recent years of the corpus, where 11 of the 14 BPMN papers were published between 2023 and 2025, contrasts with an earlier period in which UML and Blockly were the approaches of choice.

7.5.2 Temporal Evolution

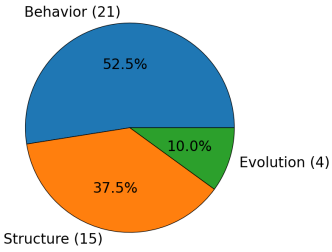
The field exhibits a clear three-phase trajectory. Between 2018 and 2020, UML and Blockly dominated publications, as depicted in Figure 7.1b, both of which are based on notations already familiar to software developers and educators. This initial phase established the foundational vocabulary for smart contract visualization, demonstrating that general-purpose formalisms could be adapted to blockchain contexts.



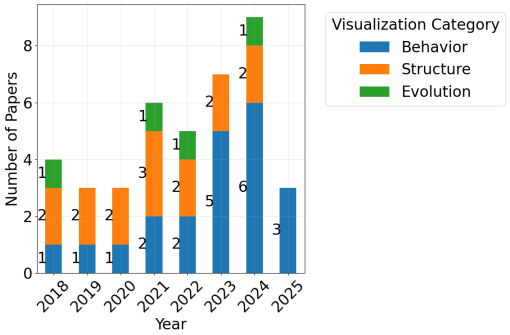
(a) Distribution of Visual Approaches



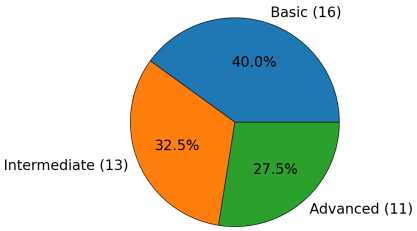
(b) Papers by Year and Visual Approach



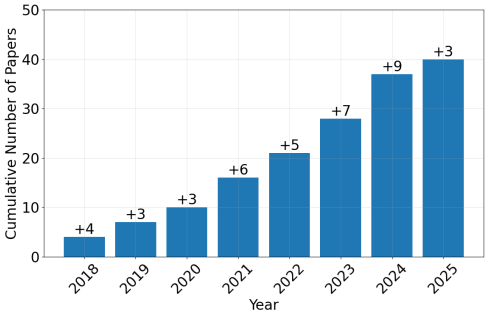
(c) Distribution of Visualization Categories



(d) Papers by Year and Visualization Category



(e) Distribution of Accessibility Requirements



(f) Cumulative Papers by Year

Fig. 7.1 Comprehensive visualization of research trends: (a) illustrates the distribution of visual approaches with BPMN (35%) being most prevalent; (b) displays visualization categories with behavior (52.5%) as the dominant focus; (c) shows accessibility requirements distribution across basic (40%), intermediate (32.5%), and advanced (27.5%) categories; (d) presents the cumulative growth of papers from 2018 to 2025; (e) details the yearly breakdown of papers by visual approach type; and (f) presents the distribution of visualization categories over time, showing an increasing focus on behavior visualization in recent years.

The period from 2021 to 2022 is characterized by methodological diversification. Researchers began experimenting with specialized representations, including EFSMs, DCR graphs, and fully custom flow-based tools, motivated by growing appreciation of the specific challenges that smart contract semantics pose for standard notations. No single formalism achieved dominance during this period, reflecting a genuine uncertainty about the most appropriate representational framework.

From 2023 onward, BPMN has emerged as the de facto standard for model-driven smart contract visualization. This convergence has several plausible explanations: BPMN's established footprint in enterprise software and its native support for multi-party interaction modeling align well with the business process automation use cases that motivate many blockchain deployments. Its normalization also facilitates interoperability between academic research tools and industry practice, a consideration that becomes increasingly important as the field matures toward production adoption. The cumulative publication count increased from 4 papers in 2018 to 40 papers by 2025 as shown in Figure 7.1f, with the steepest acceleration visible after 2021, reflecting the broader expansion of blockchain research activity during this period.

7.5.3 Visualization Category Analysis

The three visualization categories defined in Section 7.2.1 are represented in the corpus with substantially different frequencies. Behavior visualization accounts for 52.5% of papers (21 studies), structure visualization for 37.5% (15 studies), and evolution visualization for only 10% (4 studies) as shown in Figure 7.1c.

The temporal distribution within these categories reveals a disciplinary progression, as can be seen in Figure 7.1d. Early research focused predominantly on structural representation: of the 15 structure-focused studies, 11 were published between 2018 and 2021. This reflects the natural priority of establishing what smart contracts are before trying to represent how they behave. From 2022 onward, the field shifted its center of gravity toward behavioral visualization, with 16 of the 21 behavior-focused papers appearing in this latter period. This transition aligns with the growing use of blockchain for multi-party business process automation, in which the execution semantics and inter-party interaction patterns are of primary practical interest.

The scarcity of evolution-focused visualizations is notable. With only four studies (10%) that address how contracts change over time, there is a significant gap relative to the importance of this dimension in practice. As upgradeability patterns, proxy contracts, and multi-version deployments become more common, tools that visualize the evolution of contract logic and state across deployment versions will become increasingly necessary.

7.5.4 Accessibility Requirements and UX Evaluation

As can be seen in Figure 7.1e, the distribution of accessibility requirements across the corpus is broadly even: 40% of the studies (16 papers) target users with basic technical expertise, 32.5% (13 papers) address intermediate users, and 27.5% (11 papers) are designed for advanced practitioners. Block-based visualizations predominantly occupy the basic accessibility tier, consistent with their educational orientation and the prior evidence that such interfaces lower the barrier to entry for novice programmers. BPMN-based approaches tend toward intermediate and advanced levels, reflecting their use in complex business process automation scenarios requiring familiarity with the notation.

The most consequential finding of the analysis concerns the validation of the user experience. Only 8 of the 40 papers (20%) report formal UX testing. Among these, most involved small samples: 6 participants in Vieira and Vilain [135], 7 in Samanipour et al. [158], 9 in Ye et al. [149], 12 in Wen et al. [150], and 17 in Tsai et al. [151], while Gomez et al. [152] report an evaluation without stating the number of participants. The two studies with larger samples, Hdhili et al. [140] with 58 participants and Kopke et al. [101] with 42, remain the exceptions. This systematic underinvestment in user-centered evaluation suggests that smart contract visualization remains largely in a phase of technical exploration, with insufficient attention devoted to whether the proposed tools deliver measurable comprehension benefits to their intended users. A positive signal is that 6 of the 8 UX-evaluated studies were published between 2023 and 2025, indicating a growing awareness of the need to close the loop between tool design and empirical usability evidence.

7.5.5 Use Case Diversity

The 40 selected papers apply smart contract visualization to a remarkably diverse set of domains. Financial services scenarios, including voting systems, auctions, lending platforms, and insurance claims, are the most common category. Supply chain management, real estate transactions, and healthcare process automation are also well-represented. Education, document certification, and general business workflows appear in multiple studies. This breadth of application domains underscores two conclusions: first, that the demand for accessible smart contract tooling is genuinely cross-sectoral; and second, that no single visualization approach has yet demonstrated sufficient generality to serve all domains equally well. Several high-expressivity notations, such as networks of timed automata, are sufficiently powerful to represent complex temporal obligations but impose significant expert knowledge requirements, while block-based environments accessible to novices cannot easily represent the conditionality present in, for example, a mortgage contract.

7.6 Discussion

The mapping study reveals a field that has matured considerably in the breadth and sophistication of its visual proposals, but remains immature in its commitment to rigorous usability evaluation. The convergence of recent work around BPMN is encouraging from the perspective of interoperability and industrial adoption, but it should not be taken to indicate that the problem of smart contract visualization is solved. BPMN's dominance is strongest in the behavioral category and in scenarios involving multi-party collaboration; it has limited traction in the structural and evolution categories, where the internal composition of contract code and its versioned history must be represented. Filling these gaps may require adaptations of BPMN itself or the development of complementary notation that integrates with it.

The low prevalence of UX testing is the most actionable finding of this analysis. Visual tools for smart contract development are only as valuable as the comprehension gains they deliver to real users. The existing evidence base, drawn largely from studies with fewer than 20 participants and often without control conditions, is insufficient to establish which representational choices translate into genuine improvements in comprehension speed, error detection rate, or development accuracy.

Rigorous user studies, grounded in validated instruments such as the NASA Task Load Index or the System Usability Scale, and applying appropriate statistical power calculations are needed before any visualization approach can be confidently recommended for practitioner adoption. The contrast between the 20% UX evaluation rate observed here and the evaluation practice in adjacent fields such as software engineering education is striking and points to an urgent research priority.

The low representation of the evolution category is a further open problem. The production blockchain environment increasingly features complex proxy-based upgradeability patterns, multi-version contract families, and the need to audit historical state transitions for regulatory compliance. None of the reviewed visualization tools addresses these needs in a satisfactory way. The development of visual formalisms capable of representing contract lifecycles across multiple deployed versions, including the provenance of state changes and the mapping between successive code versions and their on-chain effects, constitutes a largely unoccupied research niche.

7.7 Conclusions

This chapter has presented a systematic mapping of 40 peer-reviewed works on smart contract visualization published between 2018 and 2025. The analysis reveals a clear evolutionary trajectory: the field progressed from early structure-focused work employing UML and Blockly, through a diversification phase in which custom and specialized notation proliferated, to a recent consolidation around BPMN as the preferred formalism for model-driven smart contract development. Behavioral visualizations have come to predominate over structural ones, reflecting the shift of blockchain applications toward multi-party process automation.

Three main gaps were identified. First, evolution-focused visualizations remain underrepresented despite the practical importance of contract upgradeability and state history in production deployments. Second, formal UX evaluation is absent in 80% of the reviewed works, leaving the actual comprehension benefits of proposed tools largely unvalidated. Third, no existing approach offers a unified visual language capable of addressing structural, behavioral, and evolutionary dimensions within a single coherent framework.

These gaps motivate the design and evaluation of the smart contract comprehension tool presented in Chapter 8, which integrates UML-based visualization with automated security analysis and an AI-powered annotation assistant, and subjects the resulting system to a controlled user study using the NASA Task Load Index.

Chapter 8

Making Smart Contracts Easier to Understand

8.1 Introduction

The preceding chapter surveyed the landscape of smart contract visualization and identified three consequential gaps: the near-absence of evolution-focused formalisms, the systematic underinvestment in formal usability evaluation, and the lack of any unified tool that combines structural visualization with automated security analysis and intelligent annotation. This chapter addresses the latter two gaps directly by presenting the design, implementation, and empirical user evaluation of a smart contract comprehension tool that integrates all three capabilities within a single open-source application.

The motivation for this work is grounded in the structural inequity identified throughout this thesis. Smart contracts, once deployed to the Ethereum blockchain, are immutable: vulnerabilities that survive the development phase cannot be patched in place, yet the process of auditing and understanding a contract's code requires a level of Solidity expertise that the majority of intended users—lawyers, small-business operators, and developers entering the blockchain space—do not possess. The previous work surveyed in Chapter 7 demonstrated that visualization tools targeting novice users rarely integrated security analysis capabilities, while tools that included security auditing were oriented toward expert practitioners and provided no mechanisms for AI-assisted comprehension. The present contribution closes this

gap by coupling a UML-based graphical interface with a Slither-powered security auditor [19] and an OpenAI-powered AI assistant.

The tool is built as a serverless microservice application implemented in Golang and Flutter. Its architecture decomposes the core concerns of parsing, auditing, version tracking, and AI annotation into four independent microservices, coordinated by a Backend-For-Frontend (BFF) layer. User interaction is mediated through a multi-page Flutter application available as both desktop and web software. The system was empirically evaluated against Remix IDE, the de facto standard integrated development environment for smart contract analysis, through a controlled within-subjects user study involving 36 volunteers, using the NASA Task Load Index (NASA-TLX) as a validated instrument for workload measurement [42].

The chapter is organized as follows. Section 8.2 presents the related work on low-code and no-code smart contract tools. Section 8.3 describes the architecture and implementation of its four microservices. Section 8.4 describes the client-side user interface and the visualization approach. Section 8.5 presents the experimental protocol and participant characteristics. Section 8.6 reports and interprets the NASA-TLX results. Section 8.7 concludes with a discussion of limitations and future directions.

8.2 Related Work

Since smart contracts have become widespread, researchers and practitioners have sought to reduce the technical barrier to their development and understanding. The resulting body of work can be organized along two principal axes: the visual formalism employed and the target audience addressed. Table 8.1 summarizes the most representative tools identified in the literature.

Low-code construction tools constitute the largest family, within which block-based environments are the dominant sub-group. Curty et al. [157] introduce SmartCML, a domain-specific visual modeling language for Ethereum-compatible smart contracts that generates fully functional Solidity code with type checking; it improves stakeholder communication but does not address security vulnerabilities. Gomez et al. [152] propose SmaCly, a web application that uses Blockly-based visual programming to generate Solidity code, primarily aimed at learners entering

Tool	**Formalism**	**LLM**	**Security**	**Audience**
SmartCML [157]	Graphical DSL	–	–	Novice
SmaCly [152]	Block-based	–	–	Novice
BPMN tool [147]	BPMN	–	✓	Novice
Blockly2Hook [59]	Block-based	–	–	Novice
EDALoCo [143]	Flow-based	–	–	Novice
Smart-Graph [134]	UML-based	–	–	Expert
SmartBuilder [138]	Block-based	–	–	Novice
LATTE [132]	Action graph	–	–	Novice
Kobusinska [52]	Block-based	–	–	Novice
Petrovic [97]	LLM	✓	–	Intermediate
Proposed tool	**UML**	✓	✓	**Novice**

Table 8.1 Representative low-code and no-code tools for smart contract development and analysis. The *Formalism* column groups the tools more coarsely than the systematic mapping study of Chapter 7: each label generalises the finer visual-approach classification recorded in Table 7.1, which remains the reference classification for the works surveyed there.

blockchain programming; evaluations conducted in a postgraduate course showed positive perceptions of usability, although students reported difficulty with variable declaration and initialization. Merlec et al. [138] present SmartBuilder, a Blockly-based tool targeting Hyperledger Fabric that maps visual blocks to Go source code and provides natural-language summaries. Tan et al. [132] describe LATTE, which departs from the block paradigm: contract elements are manipulated directly as the interconnected nodes of an action graph, and the tool interactively reports the gas cost of each visual action performed during construction. It generates optimized Solidity, but lacks support for advanced language features and does not integrate any security analysis. Kobusinska and Wilczynski [52] develop a Blockly-powered service that incorporates built-in validation mechanisms to prevent syntactically invalid block connections, offering some mitigation of development errors at the cost of reduced flexibility for expert users. Rosa-Bilbao et al. [143] present EDALoCo, a low-code event-driven platform built on the flow-based Node-RED environment, which integrates blockchain within a three-layer IoT architecture, demonstrated on a vaccine temperature monitoring use case.

UML-based approaches occupy a smaller but distinct niche. Pierro [134] proposes Smart-Graph, a web tool that generates augmented UML class diagrams of Ethereum smart contracts, extending standard UML with Solidity-specific constructs such as function modifiers and fallback functions; however, the tool targets experienced developers and provides no security auditing or AI-assisted annotation. Shen

et al. [147] combine BPMN with a Blockly-based code generator for multi-party scenarios and integrate Slither for security auditing, but the mapping between the BPMN model and the generated code blocks requires manual intervention when clauses exceed the system's translation capabilities.

LLM-augmented tools form an emerging and still sparse category. Petrovic and Al-Azzoni [97] demonstrate a workflow in which a domain-specific language description (iContractML 2.0) is submitted to ChatGPT for Solidity generation, reporting two case studies at a high level without formal security verification. Trestioreanu et al. [59] present Blockly2Hook, a visual programming environment for the XRP ledger that leverages Blockly to reduce the entry barrier for non-experts, with potential extension to Ethereum; the block-based abstraction limits flexibility for advanced users.

Together, this prior work reveals the gap addressed by the present tool. As shown in Table 8.1, no existing tool offers simultaneously UML-based visualization, LLM-powered annotation, and integrated security auditing for a novice target audience. The present contribution fills this position in the design space and subjects the resulting system to a controlled workload study, a validation step absent from nearly all of the related tools.

8.3 System Architecture

The tool adopts a microservice architecture implemented in Golang and Flutter, chosen for its modularity, fault tolerance, and ease of horizontal scaling. The architecture comprises four core microservices, Codec, Auditor, Database, and AI Assistant, coordinated by a Backend-for-Frontend layer that exposes a REST API to the client application. Figure 8.1 illustrates the full system.

8.3.1 Codec Microservice

The Codec microservice is responsible for transforming Solidity source code into a structured JSON representation and, conversely, for regenerating valid Solidity from a modified JSON structure. The transformation is implemented through a custom Golang library built on the ANTLR parser generator framework. The resulting

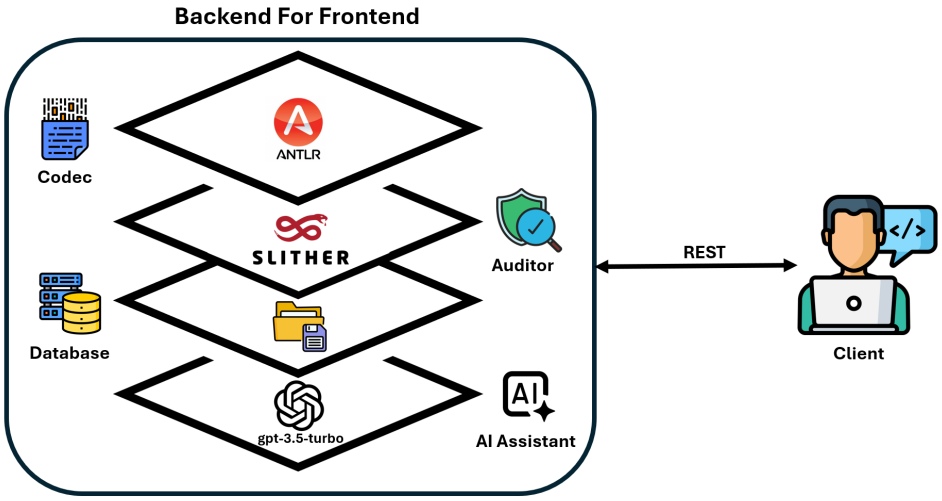


Fig. 8.1 System architecture of the smart contract comprehension tool. The four microservices (Codec, Auditor, Database, AI Assistant) are coordinated by a Backend-for-Frontend layer.

JSON encodes the full hierarchy of smart contract components, such as contracts, state variables, functions, modifiers, events, constructors, and their respective bodies and signatures, in a form that can be consumed by the visualization layer and the AI Assistant. The bidirectional nature of the codec is essential for the editing workflow: users can modify individual component bodies through the graphical interface, and the updated JSON is re-encoded into syntactically valid Solidity before being forwarded to the Auditor.

8.3.2 Auditor Microservice

The Auditor microservice wraps Slither [19], the static analysis framework described in Section 2.2.4, to provide automated vulnerability detection. As established in previous chapters, Slither operates by converting Solidity source code into the SlithIR intermediate representation and applying a library of detectors corresponding to known vulnerability patterns, producing findings classified by severity as high, medium, or low. The Auditor receives Solidity source code from the BFF, invokes Slither, parses the resulting vulnerability report, and returns structured findings to the client. This design decouples the analysis engine from the presentation layer, permitting future substitution of alternative analyzers or the addition of supplementary tools alongside Slither.

8.3.3 Database Microservice

A lightweight Database microservice provides version control functionality, recording successive user modifications to the smart contract as distinct revisions. This component gives users the ability to compare the original imported contract with edited versions and to revert modifications incrementally, supporting an iterative review workflow without requiring external version control tooling. The service exposes a minimal REST interface and persists revisions in a structured format compatible with the Codec's JSON schema.

8.3.4 AI Assistant Microservice

The AI Assistant microservice leverages OpenAI's `gpt-3.5-turbo` model to automate three annotation tasks: generating natural-language descriptions of individual smart contract components, establishing and labeling functional relationships between components, and identifying potential improvements to function implementations. The assistant receives the structured JSON produced by the Codec and enriches each component with contextual annotations using the model's function-calling capability, which ensures that responses are returned in the structured format expected by the visualization layer. The primary purpose of this microservice is to render the abstract representation of the contract interpretable to users without Solidity expertise, bridging the gap between code-level constructs and their real-world semantics.

8.4 User Interface and Visualization

The client application is developed in Flutter, and it supports both desktop and web deployment. It is organized into three pages: a welcome page for contract upload and analysis summary, a contract page for side-by-side code and AI-generated description, and a settings page for configuration. Figures 8.2, 8.3, and 8.4 illustrate the three primary interaction surfaces.

On the welcome page, the user uploads a Solidity source file via a REST call to the BFF. The returned JSON is parsed into entities, such as vulnerabilities and the contract component hierarchy, presented through a tab-based interface. Vulnerability

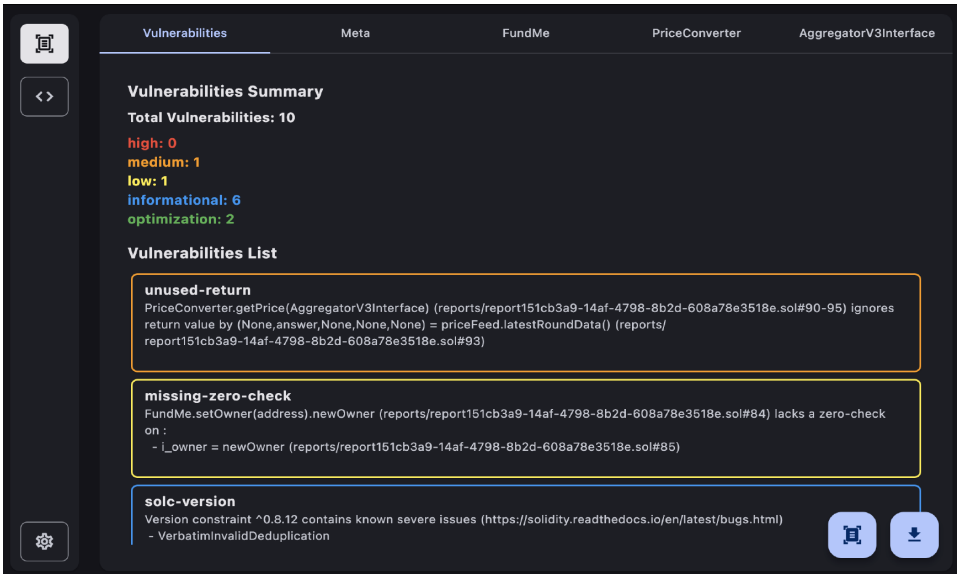


Fig. 8.2 The welcome page showing the smart contract analysis overview, with vulnerability findings grouped by severity and a tab-based navigation of contract components.

findings are grouped by severity level, giving users an immediate risk overview without requiring familiarity with Slither's command-line output format.

The contract page presents the original Solidity source code along with the AI-generated natural-language descriptions produced by the AI Assistant. This side-by-side layout enables users who are unfamiliar with Solidity syntax to correlate specific code segments with their functional purpose. Contracts embedded within the same source file are accessible via distinct tabs, preserving navigational clarity in multi-contract files.

The central visualization is a custom Flutter widget that renders smart contract components as UML-inspired cards arranged on an interactive grid. Each card represents a contract element, such as a function, modifier, state variable, event, or constructor, and displays the AI-generated annotation alongside a compact representation of the component's signature. The connection lines between the cards convey parental and functional relationships, dynamically updated as the user repositions elements within the grid. Selecting a card opens a side panel that displays the full component body (i.e., the raw Solidity code) together with its metadata, and allows inline editing of the body without requiring direct interaction with the source text. This editing mechanism is deliberately constrained to the component level, reducing the risk of inadvertent structural corruption while enabling targeted modifications.



Fig. 8.3 The contract page displaying source code alongside AI-generated natural-language descriptions of functions, enabling users to correlate code segments with their real-world purpose.

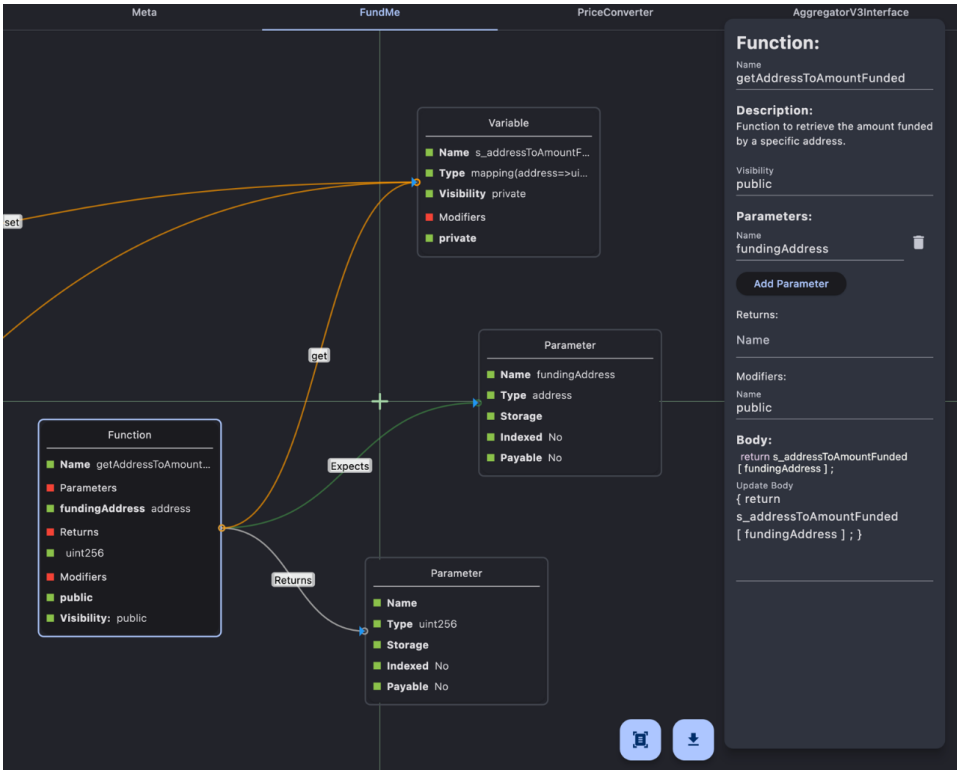


Fig. 8.4 The interactive UML-based grid view of smart contract components, with connection lines representing functional and parental relationships and a side panel providing component-level details and inline editing.

The grid layout addresses the challenge of information density identified in the Smart-Graph evaluation [134], where complex contracts produced visually crowded diagrams; by distributing components across a navigable spatial canvas rather than a static diagram, the widget preserves detail without sacrificing legibility.

8.5 Experimental Evaluation

8.5.1 Study Design

The evaluation used a within-subjects crossover design in which each participant performed a set of smart contract analysis tasks using either the proposed tool or Remix IDE, then switched to the other environment for a second set of tasks. Participants were randomly assigned to one of two groups: Group A analyzed the first set of contracts with the proposed tool and the second set with Remix IDE, while Group B followed the reverse order. Assignment to groups is therefore between subjects, but every participant is measured under both tools. This counterbalanced design controls for contract difficulty and order effects, and because each participant contributes one observation per tool, the resulting NASA-TLX scores form matched pairs, which is the condition required by the Wilcoxon signed-rank analysis adopted below.

Remix IDE was selected as the baseline for comparison because of its status as the most widely adopted integrated environment for Ethereum smart contract development, combining open-source accessibility, a rich plugin ecosystem, built-in compilation and debugging capabilities, and broad adoption in both academic research and industry practice. Its feature set represents the best conventional alternative available for users without dedicated visualization tools.

The primary measurement instrument was the NASA Task Load Index (NASA-TLX) [42], a multidimensional workload scale introduced by Hart and Staveland in 1988 that quantifies perceived workload in six dimensions: Mental Demand, Physical Demand, Temporal Demand, Performance, Effort, and Frustration. Each dimension is rated on a 100-point scale with 5-point increments. The instrument produces both dimension-level scores and a weighted composite score, where the weights are derived from 15 pairwise comparisons performed by each participant. NASA-TLX has accumulated an extensive validation literature in domains spanning aviation,

healthcare, software development, and human-computer interaction [42], making it an appropriate choice for assessing the cognitive demands of smart contract analysis tasks. The determination of the sample size followed the convention established for Wilcoxon signed-rank analysis: a minimum of 35 subjects are required to achieve a statistical power of 0.80 at an effect size of 0.5, leading to the recruitment of 36 participants.

A time limit of five minutes per task was established to ensure the comparability of temporal demand measurements among participants and to prevent task-completion variability from confounding workload scores.

8.5.2 Participants and Protocol

Thirty-six volunteers employed in IT-related companies participated in the study. Participants reported a mean programming expertise of 3.0 on a five-point Likert scale and a mean blockchain expertise of 2.0 on the same scale, confirming the target profile of technically literate but blockchain-inexperienced users. All survey responses and smart contract sets are publicly available in the project repository.[1]

The experimental protocol comprised three sequential phases. An introductory session provided foundational knowledge about blockchain technology and smart contracts, clarified the objectives of the evaluation, and explained the operation of both tools. A training session then familiarized the participants with the proposed tool and with Remix IDE through guided demonstrations. Finally, participants independently performed the analysis tasks on the two sets of contracts, completing the NASA-TLX survey immediately after each session. Three of the four tasks required participants to identify specific properties of a contract (e.g., identifying the access control policy on a given function or locating the vulnerability class reported for a given line); the fourth required a light editing operation.

[1] https://github.com/GangemiLorenzo/MasterDegreeThesis

8.6 Results

8.6.1 Task Completion Time

Participants completed three of the four analysis tasks in less time using the proposed tool than with Remix IDE. The fourth task, which required a more complex editing operation, showed a marginal time advantage for Remix IDE, likely attributable to the richer code-editing capabilities of a dedicated IDE relative to the tool's component-level inline editor. These timing differences, however, do not reach statistical significance: the per-task comparisons yield p-values between 0.06 and 0.30, none below the conventional 0.05 threshold. This is expected from the structure of the protocol, since any given contract was analysed with only one of the two tools by each participant, so the task-time comparison is made between two groups of 18 rather than within participants and does not benefit from the pairing available to the workload analysis. The timing results are therefore consistent with, but not on their own confirmatory of, the hypothesis that the proposed tool accelerates information-retrieval tasks through its structured visualization; the workload measurements reported below carry the inferential weight of the evaluation.

8.6.2 NASA-TLX Workload Results

Figure 8.5 reports the raw scores, dimension weights, and composite workload scores for both tools.

The composite NASA-TLX score was 52.66 for the proposed tool and 71.74 for Remix IDE, a difference of approximately 19 points that represents a reduction of 27% in the overall perceived workload. This is a practically significant margin: prior NASA-TLX studies in software engineering contexts typically classify differences exceeding 10 points as meaningfully impactful on user experience.

The reduction is also statistically significant. Because every participant completed the NASA-TLX instrument once per tool, the scores form matched pairs, and a two-sided Wilcoxon signed-rank test on the participant-level weighted scores rejects the null hypothesis of no difference ($n = 36$, $W = 1$, $p < 0.001$).[2] Thirty-

[2]The composite values quoted above, 52.66 and 71.74, apply the dimension weights aggregated over the whole sample. A paired test instead requires one score per participant, obtained by weighting

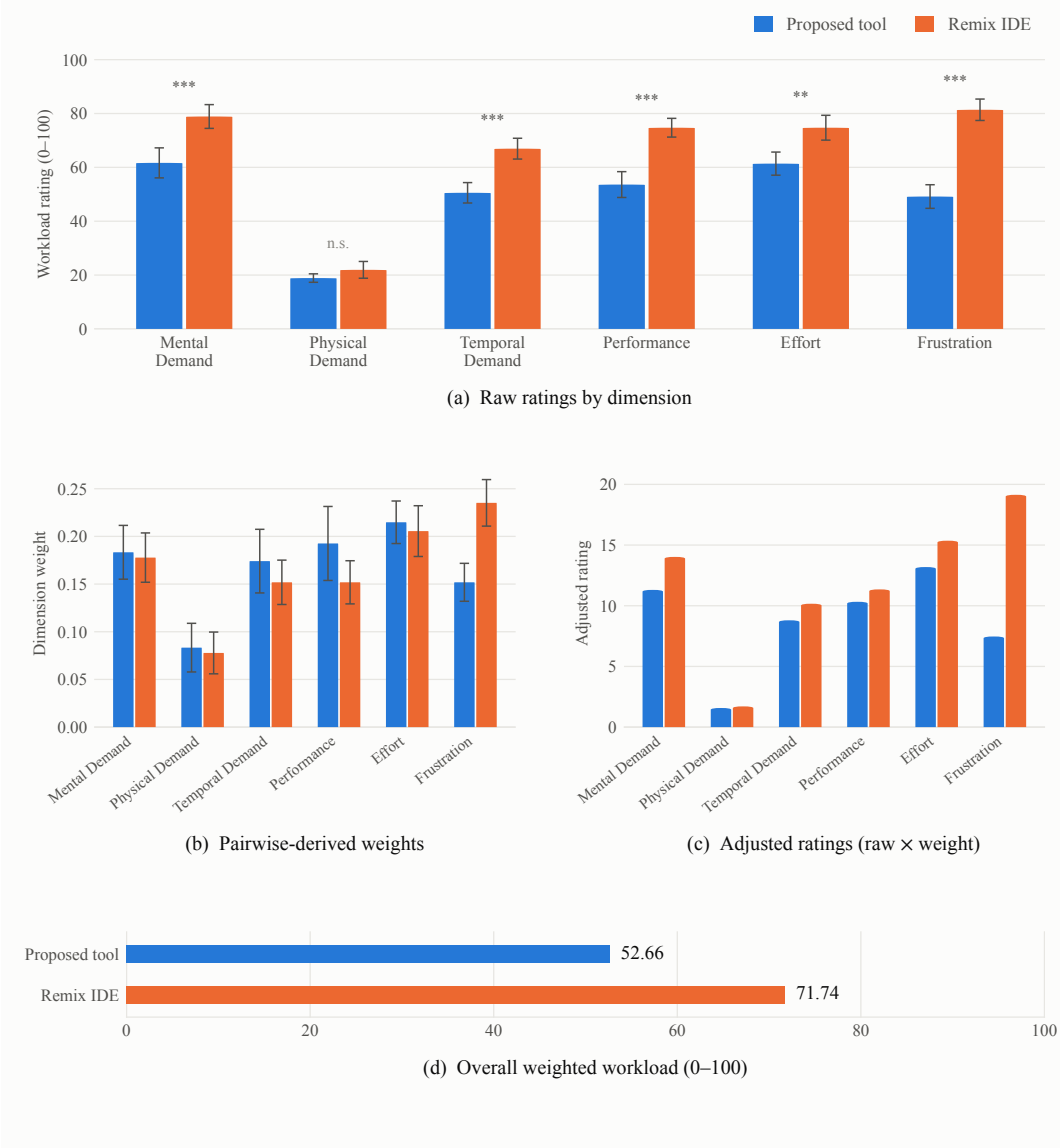


Fig. 8.5 NASA-TLX results for the proposed tool (blue) and Remix IDE (orange), computed from the 36 matched responses. (a) Raw ratings per dimension, with 95% confidence intervals of the mean and the significance of the per-dimension Wilcoxon signed-rank tests after Holm correction ($^{***}p < 0.001$, $^{**}p < 0.01$, n.s. not significant); (b) dimension weights derived from the 15 pairwise comparisons; (c) adjusted ratings, the product of each raw rating and its sample-level weight, which sum to the composite; (d) the weighted overall workload. Lower scores indicate lower perceived workload.

five of the 36 participants reported a lower workload with the proposed tool. The Hodges–Lehmann estimate of the median reduction is 17.3 points, with an exact 95% confidence interval of $[13.7, 20.7]$, and the matched-pairs rank-biserial correlation of 0.997 indicates a large effect. The conclusion is not an artefact of the weighting scheme: applying the same test to the unweighted Raw-TLX average gives $W = 0$ and $p < 0.001$, with all 36 participants scoring the proposed tool lower.

Examining individual dimensions reveals a different pattern. Mental Demand was rated 61.67 for the proposed tool versus 78.89 for Remix IDE, indicating that AI-generated annotations and structured component hierarchy reduced the cognitive effort required to comprehend contract semantics. Temporal Demand showed a 16-point gap (50.56 versus 66.94), corroborating the task-completion time advantage observed for three of the four tasks. The Performance dimension favored the proposed tool by approximately 21 points (53.61 versus 74.72), and participants reported greater confidence in their task results when using the visualization interface. Effort was approximately 13 points lower for the proposed tool (61.39 versus 74.72), consistent with the observation that Remix IDE requires trial-and-error exploration of code in the absence of structural guidance. Physical Demand was uniformly low for both tools (18.89 versus 21.94), as expected for screen-based tasks without physical interaction requirement; it is the only dimension on which the two tools do not differ significantly.

The most pronounced difference appeared in Frustration, where Remix IDE scored 81.39 against 49.17 for the proposed tool, a gap of 32 percentage points. This result is notable because frustration is typically the dimension most sensitive to the mismatch between user mental models and system affordances: a highly capable tool whose operation is opaque to novice users produces frustration scores disproportionate to its technical merit. The substantially lower frustration score of the proposed tool suggests that the combination of visual abstraction, AI annotation, and structured navigation was substantially aligned with participants' expectations and mental models of smart contract analysis.

---

each participant's ratings with that participant's own 15 pairwise comparisons; this yields sample means of 54.07 and 71.50, and it is these participant-level scores that the test is applied to. The exact two-sided p-value is 5.8×10^{-11}. The composite differs by 0.04 from the 52.62 reported in the underlying publication: one of the 540 pairwise-comparison cells in the published dataset carries a corrupted label and was therefore left uncounted in the original tally. Restoring it changes no test result, and all figures reported here are computed from the corrected data.

The weighted composite scores, which account for each participant's individual dimension priorities, confirm the direction and magnitude of these findings: the adjusted Frustration score was 7.47 for the proposed tool against 19.14 for Remix IDE, reinforcing the conclusion that reducing cognitive friction in the comprehension workflow is the mechanism through which the tool delivers its primary benefit.

Table 8.2 summarizes all dimension scores.

Dimension	**Proposed Tool**	**Remix IDE**	p
Mental Demand	61.67	78.89	< 0.001
Physical Demand	18.89	21.94	0.074
Temporal Demand	50.56	66.94	< 0.001
Performance	53.61	74.72	< 0.001
Effort	61.39	74.72	0.002
Frustration	49.17	81.39	< 0.001
Overall (weighted)	**52.66**	**71.74**	< 0.001

Table 8.2 NASA-TLX raw ratings per dimension and weighted overall workload score for the proposed tool and Remix IDE. Lower values indicate lower perceived workload. The p-values are those of two-sided Wilcoxon signed-rank tests on the 36 matched pairs, Holm-corrected across the six dimensions; the overall row reports the test on the participant-level weighted composite.

8.6.3 Discussion

The results support three conclusions. First, UML-based visualization of smart contract components, enriched with AI-generated natural-language annotations, measurably reduces the cognitive load associated with contract comprehension for users with limited Solidity experience. The 27% reduction in composite workload and the 32-point reduction in Frustration demonstrate that structural visualization is not only esthetically preferable to raw source code inspection, it delivers quantifiable usability gains.

Second, the gap between the two tools is most pronounced along dimensions that reflect the mismatch between user mental models and tool affordances: Mental Demand, Frustration, and Performance. This suggests that the primary mechanism of benefit is not speed per se, but the reduction of cognitive friction in locating and interpreting contract semantics. The AI Assistant, which translates opaque function implementations into natural language, is likely the dominant contributor

to the Mental Demand and Frustration advantages, while the grid-based navigation accounts for the Temporal Demand improvement.

Third, the time disadvantage observed on the fourth task, the only task requiring substantive code editing, points to the tool's principal limitation relative to Remix IDE. The component-level inline editor, designed to prevent structural errors by constraining modifications to individual bodies, sacrifices the flexibility and keyboard efficiency of a full-featured code editor. This trade-off is appropriate for the tool's target audience of novice users performing comprehension and light auditing tasks, but it represents a constraint that future iterations should relax for users with higher programming proficiency.

These findings complement the broader quantitative picture assembled in Chapters 3–6: while those chapters examined the capacity of LLM to generate and correct smart contracts, the present chapter demonstrates that LLM-assisted annotation, even using a relatively modest model (`gpt-3.5-turbo`), contributes meaningfully to the comprehension of contracts produced by any automated or manual workflow. The tool thus occupies a distinct and complementary role within the thesis's research program: it is not a code generator but a comprehension accelerator, applicable to any Solidity contract regardless of its provenance.

8.7 Conclusions

This chapter presented a smart contract comprehension tool that integrates UML-based visualization, Slither-powered security auditing, and GPT-assisted annotation in a serverless microservice architecture. The system was designed to serve users with limited Solidity expertise and was evaluated through a within-subjects user study with 36 participants using the NASA-TLX workload instrument.

The evaluation demonstrated a statistically and practically significant reduction in perceived workload compared to Remix IDE (Wilcoxon signed-rank, $n = 36$, $p < 0.001$), with a composite score of 52.66 versus 71.74 and particularly pronounced improvements in Frustration (49.17 versus 81.39) and Mental Demand (61.67 versus 78.89). Participants completed three of four analysis tasks more efficiently with the proposed tool, although these timing differences did not themselves

reach significance, and the structured visualization approach was well-aligned with the mental models of technically literate but blockchain-inexperienced users.

Several directions for future work are indicated by the results and by the limitations of the current implementation. Extending the AI Assistant with a knowledge base of smart contract-specific patterns, vulnerability semantics, and common Solidity idioms would improve annotation precision beyond what a general-purpose language model can deliver. Developing a contract documentation generation subsystem that produces comprehensive inline explanations, usage examples, and best-practice notes would extend the tool's utility to contract maintenance workflows. Integrating a gas optimization advisor and a formal testing framework for automated coverage analysis would broaden the applicability of the tool to production deployment contexts. Finally, the user study should be extended to include non-IT users, legal professionals, and business analysts, whose comprehension challenges differ qualitatively from those of IT-trained participants, and whose inclusion would provide a more complete empirical basis for the claim that visualization lowers the entry barrier for all categories of smart contract stakeholder.

Chapter 9

Conclusions

9.1 Overview

This thesis set out to investigate whether, and to what extent, Large Language Models can reduce the gap between the technical complexity of smart contract development and the broad population of practitioners who stand to benefit from blockchain automation. The research program was motivated by a structural tension at the core of the Ethereum ecosystem: a technology with transformative potential for decentralized finance, legal automation, and supply chain governance, yet one that remains locked behind a programming discipline inaccessible to most of its intended beneficiaries. The six contributions reported in Chapters 3 through 8 addressed this challenge through three complementary research questions, concerning vulnerability correction, automated contract generation, and visual comprehension, and together they constitute a coherent empirical response to the central research problem.

The remainder of this chapter synthesizes the main findings of each contribution, draws cross-cutting conclusions about the state of LLM-assisted smart contract engineering, identifies the principal limitations of the work, and outlines the directions that appear most promising for future research.

9.2 Summary of Contributions

9.2.1 LLM-Based Vulnerability Correction

Chapter 3 addressed the question of whether a general-purpose LLM, specifically `gpt-3.5-turbo` operating as the public ChatGPT interface, could automatically correct Solidity security vulnerabilities identified by Slither in the SmartBugs-Curated benchmark [84]. The study introduced a structured iterative pipeline in which Slither detected vulnerabilities and ChatGPT was prompted to correct them over four successive rounds, both with and without a structured bug description derived from Slither output.

The headline result was that ChatGPT returned compilable Solidity code in approximately 89.7% of all API calls, a baseline syntactic fidelity that distinguishes it from naive code-generation approaches and confirms its suitability for integration into automated pipelines. In the best round of four attempts, the model corrected at least one Slither-detected vulnerability in 57.1% of contracts operating in code-only mode and in 58.5% of contracts when provided with a structured Slither bug description, representing a modest but consistent $+1.4$ percentage-point improvement attributable to structured contextual input. Complete remediation of all detected vulnerabilities was achieved in 27.1% of cases in code-only mode. The model showed the strongest performance on Medium-severity vulnerabilities, reaching correction rates of 70–72% in the best round, while High-severity (approximately 49%) and Low-severity (45–50%) vulnerabilities proved more resistant.

These results establish a practical baseline for automated LLM-driven vulnerability management: ChatGPT can function as a viable first-pass patch generator within a Slither-in, Slither-out pipeline, reducing the cognitive burden on developers to a final review step. At the same time, the study identified the fundamental ceiling imposed by purely automated evaluation: static analysis provides no signal about the semantic correctness of the generated patch, and a contract that passes Slither may still be logically unsound. This limitation motivated the introduction of expert manual evaluation frameworks in subsequent contributions.

9.2.2 Benchmarking LLM-Based Smart Contract Generation

Chapter 4 introduced a reusable pipeline for the automatic generation of Solidity smart contracts from legal agreements using GPT-4, structured according to the CO-STAR prompt engineering methodology [38]. The contribution included both the generation pipeline and a multi-metric benchmarking suite, the Weighted Total Score (WTS) framework, comprising seven automated indicators: compilability, vulnerability density, Solidity version compliance, number of generated contracts and functions, comment coverage, and initialized parameter rate.

Evaluated on a property rental agreement, the pipeline achieved a compilability rate of 98.1% across the six selected prompt templates (54 generations) and demonstrated that CO-STAR structured prompts with explicit Objective and Requirements dimensions consistently outperformed underspecified prompts in both average quality and inter-invocation reproducibility. Mean WTS showed no systematic trend across temperature values in the practical range $[0.0, 1.0]$, an observation with direct implications for deployment: users can rely on the default temperature setting without evidence of a quality penalty. These results established that the pipeline is a practical instrument for non-technical users seeking to automate the translation of legal agreements into executable code.

The chapter simultaneously exposed the ceiling of purely automated evaluation. No metric within the WTS framework was capable of detecting logical flaws in the generated contract logic, incomplete clause translation, or semantic deviations from the input document. Exposing and quantifying this gap between automated metrics and semantic correctness became the central research question in Chapter 5.

9.2.3 The Limits of Automated Evaluation: Expert Manual Assessment

Chapter 5 deepened the analysis of LLM-based generation through a comprehensive expert manual evaluation of 68 Solidity contracts produced by GPT-4 from a property rental agreement [30]. The evaluation used a structured manual inspection protocol organized along three dimensions: the presence, completeness, and logical correctness of each expected function; the fidelity of parameter initialization to the

values specified in the source agreement; and the quality and informativeness of inline code comments.

Manual inspection identified five categories of systematic failure that were entirely invisible to automated tools: funds permanently frozen in the rent payment logic due to the absence of a landlord withdrawal function; absent or incorrectly implemented security deposit collection; widespread omission of late-fee enforcement logic; legally inadequate lease termination conditions; and universal currency mismanagement, in which the US dollar amounts of the agreement were carried over unchanged as Ether quantities, so that a monthly rent of $850 USD becomes 850 Ether, with no contract in the corpus performing a USD-to-ETH conversion. The central conclusion was unambiguous: GPT-4, even when prompted according to the structured CO-STAR methodology, is incapable of generating production-ready smart contracts for agreements of real-world legal complexity. The WTS is a necessary but insufficient indicator of production readiness. Expert review is not a desirable supplement to automated evaluation; it is a required component of any credible quality assessment for LLM-generated smart contracts.

This finding constitutes a methodological contribution of independent significance. It establishes that the evaluation protocols inherited from general code-generation benchmarks, such as compilability, test-suite pass rates, and static analysis scores, are systematically inadequate for the smart contract domain, where correctness must be assessed against the semantic intent of a legal document rather than against executable test cases or syntactic well-formedness criteria.

9.2.4 Comparative Multi-Model Evaluation

Chapter 6 extended the analysis to a comprehensive comparative study of four frontier commercial LLMs, more specifically `claude-3-5-sonnet-latest`, `gpt-4-turbo`, `gemini-1.5-pro`, and `mistral-large-latest`, evaluated across five legal agreements of increasing complexity (Cleaning Service, Compensation, Equipment Rental, Mortgage Note, and Property Sale) using six prompt templates and four independent runs per configuration, for a total of 480 API calls [1]. The evaluation combined automated Slither analysis with the manual pattern-based scoring framework introduced in Section 6.3.3, derived from the smart contract design pattern taxonomy of Barto-

letti and Pompianu [39] and Khan et al. [40] and extended with a domain-specific Business Logic pattern.

The study established a clear performance hierarchy, with Claude and GPT-4-Turbo sharing an identical compilability rate of 88.3% (106 of 120 contracts) and consistently outperforming Gemini (53.3%) and Mistral (29.2%) across all metrics. Performance degradation across models is non-linear with respect to agreement complexity: the Cleaning Service and Compensation agreements are handled adequately even by weaker models, while the Mortgage Note and Property Sale agreements cause substantial failures across the entire model family, with even top-performing models showing compilability rates as low as 25–50% on certain prompt configurations.

An important secondary finding concerns the apparent security advantage of lower-performing models: Gemini and Mistral exhibit fewer absolute Slither-detected vulnerabilities, but this is an artifact of their lower compilability rather than evidence of superior security practices, since Slither can only analyze contracts that compile. Among compilable contracts, the prevalence of vulnerability is comparable between models and is correlated with the complexity of agreement rather than the identity of the model. Supplying a worked exemplar together with reasoning-mode inference showed considerable promise, producing a qualitatively stronger contract even for the weakest model on the most difficult agreement. That preliminary experiment varied both factors simultaneously and therefore does not establish which is responsible, but it suggests that exemplar-based and multi-step reasoning strategies represent a productive direction for future prompt engineering research.

9.2.5 Systematic Mapping of Smart Contract Visualization Research

Chapter 7 contributed a systematic mapping study of 40 peer-reviewed works on smart contract visualization published between 2018 and 2025 [34]. The study characterized the field in five dimensions: visualization methodology, representation type (behavioral, structural, evolutionary), accessibility level, use case domain, and evaluation rigor.

Three main findings emerged. First, the field has undergone a clear evolutionary trajectory from early UML-based and Blockly-based structural representations

toward BPMN as the dominant formalism for model-driven, multi-party process automation. Second, behavioral visualizations have come to predominate over structural ones, reflecting the maturation of blockchain applications toward complex workflow automation. Third, and most actionably, 80% of the reviewed works lack any formal user experience evaluation, leaving the comprehension benefits claimed for the proposed tools largely unvalidated. These findings identified a critical gap that the contribution of Chapter 8 was explicitly designed to address: the design and user evaluation of a smart contract comprehension tool grounded in established human-computer interaction instruments.

9.2.6 A Smart Contract Comprehension Tool with User Evaluation

Chapter 8 presented the design, implementation and controlled evaluation of an open-source smart contract comprehension tool that integrates UML-based structural visualization, automated Slither security auditing, GPT-assisted annotation, and a version control subsystem for contract evolution tracking [41]. The system was implemented as a serverless microservice architecture in Golang and Flutter, deployed as desktop and web software.

The tool was evaluated through a within-subjects user study with 36 volunteers using the NASA Task Load Index (NASA-TLX) [42] as the primary instrument. The composite NASA-TLX score was 52.66 for the proposed tool versus 71.74 for Remix IDE, representing a reduction of approximately 27% in perceived workload, a difference that is statistically significant under a two-sided Wilcoxon signed-rank test on the 36 matched pairs ($p < 0.001$) and that substantially exceeded the 10-point threshold conventionally considered practically significant in software engineering contexts. The most pronounced improvements appeared in Frustration (49.17 versus 81.39) and Mental Demand (61.67 versus 78.89), indicating that the structured visualization approach is particularly effective in reducing the cognitive burden associated with interpreting unfamiliar contract code. Participants completed three of four analysis tasks more efficiently with the proposed tool.

This contribution directly addresses the gap identified in the systematic mapping study: it provides a controlled, statistically powered user evaluation of a smart contract visualization tool and demonstrates that the combination of UML-based

representation, security audit integration, and AI-assisted explanation yields a measurable and practically meaningful reduction in user workload.

9.3 Cross-Cutting Conclusions

The findings in all six contributions converge on a number of conclusions that extend beyond the scope of any individual chapter.

The sufficiency gap between automated and semantic evaluation. The most consistent finding of this thesis is that the metrics inherited from general software engineering, such as compilability, static analysis pass rates, and syntactic well-formedness, are necessary but not sufficient to evaluate LLM-generated smart contracts. The empirical evidence for this conclusion is direct: GPT-4 achieves near-perfect compilability (98.1%) and acceptable WTS scores in contracts that are nevertheless functionally broken, financially hazardous, and legally inadequate. This gap between syntactic correctness and semantic soundness is not incidental; it is structural. Solidity's type system and compiler do not enforce the business logic of the agreement they are meant to implement, and static analysis tools such as Slither are designed to detect known vulnerability patterns, not to verify specification compliance. Future evaluation protocols for LLM-generated smart contracts must incorporate semantic correctness checks grounded in the source legal documents.

The role of prompt engineering. Across all generation experiments, structured prompt engineering, particularly the CO-STAR framework, yielded consistent improvements over under-specified prompts in both average quality and reproducibility. However, the magnitude of this improvement is bounded: no prompt design tested was sufficient to overcome the fundamental difficulty of translating complex legal structures into syntactically correct, semantically complete, and security-sound Solidity code. Prompt engineering is a necessary but not sufficient instrument. Exemplar-guided prompting combined with reasoning-mode inference shows promise as a complementary strategy, particularly for weaker models on more demanding agreements, and warrants systematic investigation in future work.

LLMs as tools within supervised workflows. The consistent conclusion across all generation experiments is that current frontier LLMs, including the best-performing models in the comparative evaluation, are not yet suitable for unsupervised smart contract generation from real-world legal agreements. However, they are demonstrably useful within supervised workflows that combine LLM-assisted drafting with mandatory expert review, structured security auditing via tools such as Slither, and iterative prompt refinement. The appropriate framing for LLM-based contract generation is therefore not autonomous code generation but assisted drafting, in which the LLM handles syntactic scaffold construction, and the domain expert provides semantic validation. This framing is consistent with wider assessments of the role of LLMs in professional software engineering practice [26].

Visualization as an independent accessibility lever. The visualization contribution demonstrates that the accessibility gap in smart contract development is not exclusively a generation problem: comprehension is an equally significant barrier for the non-expert users who interact with deployed contracts without having written them. The 27% reduction in NASA-TLX workload achieved by the proposed tool establishes that structured AI-assisted visualization constitutes an independent and effective approach to lowering this barrier. The finding that visualization improvements are most pronounced in Frustration and Mental Demand scores suggests that the benefit is primarily cognitive: the tool reduces the interpretive effort required to understand contract structure and security implications, rather than simply speeding up mechanical lookup tasks.

9.4 Limitations

The contributions of this thesis are subject to several limitations that should be acknowledged.

The vulnerability correction study (Chapter 3) used `gpt-3.5-turbo` as it was the most capable publicly available engine at the time of the study (April 2023). The rapid evolution of the model since that date means that the quantitative results are best interpreted as a characterization of the capability baseline at a specific moment rather than as an absolute ceiling for LLM-based vulnerability correction.

The SmartBugs-Curated benchmark [84], while widely used in the literature, contains contracts specifically collected for vulnerability detection research and may not be representative of the distribution of vulnerabilities present in production deployments. Evaluation on a larger, more representative sample of deployed contracts would strengthen the generalizability of the findings.

The manual evaluation by experts in Chapter 5 was conducted on a single type of legal agreement (property rental) and with a single LLM. The generalizability of the specific failure modes identified, such as frozen funds, currency mismanagement, incomplete termination logic, to other types and models of agreement is an empirical question that the comparative study in Chapter 6 begins to address but does not resolve completely.

The systematic mapping study (Chapter 7) is necessarily a snapshot of a rapidly evolving field. The search protocol was applied to papers published up to mid-2025, and the trajectory of the field, particularly the consolidation around BPMN and the growing interest in AI-assisted visualization, suggests that the landscape will have evolved substantially within a short period.

The user study in Chapter 8 was conducted with 36 IT-trained participants, a constraint imposed by the statistical power requirements of the Wilcoxon signed-rank analysis [42]. The study does not include non-IT users, legal professionals, or business analysts whose comprehension challenges may differ qualitatively from those of the recruited population. The claim that visualization lowers the entry barrier for all categories of smart contract stakeholder therefore rests on an inference beyond the available evidence.

9.5 Future Research Directions

The findings of this thesis, and the limitations just described, point to several directions that warrant investigation in future work.

Domain-specific fine-tuning and retrieval-augmented generation. The most pressing near-term opportunity is the development of Solidity-specialized LLMs, trained or fine-tuned on curated corpora of high-quality, expert-validated smart contracts. Recent work has demonstrated that domain-specific fine-tuning can

substantially improve both compilation success rates and security properties in generated code [160]. A complementary strategy is retrieval-augmented generation (RAG), which augments LLM prompts at inference time with semantically relevant contract fragments retrieved from a verified knowledge base [161]. Both approaches address a common root cause identified in this thesis: general-purpose LLMs trained on broad code corpora have limited exposure to Solidity's domain-specific semantics, execution model, and vulnerability taxonomy.

Intermediate representation-guided generation. The results of Chapter 6 suggest that direct legal-to-code translation places a cognitive load on the LLM that current models cannot reliably support for complex agreements. Architectures that decompose the generation task into a sequence of intermediate representations, for instance, generating a formal process model or a finite state machine from the legal text before synthesizing Solidity code from that intermediate representation [160], may substantially improve both compilability and semantic completeness. This requirements-to-model-to-code paradigm aligns with established software engineering practice, where formal specification precedes implementation, and constitutes a promising design principle for the next generation of generation pipelines.

Automated semantic correctness verification. The manual evaluation framework introduced in Chapter 5 demonstrates the need for semantic review, but does not scale to the throughput required for continuous integration workflows. The development of automated semantic correctness checks that are grounded in formal specification languages, property-based testing, or LLM-assisted specification verification that go beyond static analysis to assess whether a generated contract faithfully implements the intent of its source document is a high-priority research challenge. Recent work on LLM-driven formal property generation for smart contracts [162] suggests that this direction is technically feasible and operationally relevant.

Evolution-aware and multi-version visualization. The systematic mapping study identified the absence of evolution-focused visualization tools as the most significant structural gap in the field. Production blockchain deployments increasingly feature complex proxy-based upgradeability patterns and multi-version contract families whose state history must be auditable for regulatory compliance. The development

of visual formalisms capable of representing contract lifecycles across multiple deployed versions, including the provenance of state changes and the correspondence between successive code versions and their on-chain effects, constitutes a largely unoccupied research niche that the comprehension tool of Chapter 8 could be extended to address.

Broader user populations and longitudinal evaluation. The user study of Chapter 8 should be extended to include non-IT users, legal professionals, and business analysts. This extension would not merely increase the ecological validity of the reported results; it would address the fundamental normative goal of the B4A project, making blockchain technology accessible to all, by providing evidence about the specific barriers and cognitive requirements of the non-technical population for whom the tool is ultimately intended. Longitudinal studies that track the development of users' mental models over extended interaction with the tool would complement the present cross-sectional snapshot evaluation and provide a richer picture of the comprehension gains attributable to visualization.

Multi-agent and ensemble architectures. The findings of this thesis suggest that no single LLM, operating alone, can reliably produce production-ready smart contracts. Multi-agent architectures, in which multiple models contribute complementary capabilities to a shared generation pipeline, such as one model drafting contract structure, another performing security review, and a third verifying semantic alignment with the source document, offer a natural design response to this limitation. Initial evidence from ensemble-based vulnerability detection systems [160] suggests that model diversity improves coverage and reduces individual model blind spots, a principle that can be transferred to generation tasks.

9.6 Closing Remarks

The research program reported in this thesis was motivated by a conviction that the gap between the technical complexity of smart contract engineering and the practical needs of non-expert users is not a fixed feature of the technology, but a contingent limitation that systematic research can reduce. The six contributions presented here provide empirical evidence that this reduction is achievable: LLMs can correct a

majority of automatically detected vulnerabilities in a structured pipeline; they can generate syntactically correct contracts from legal agreements with near-perfect reliability; visual tools grounded in established HCI instruments can reduce the cognitive workload of contract comprehension by more than a quarter; and the combination of structured prompt engineering, expert review, and automated security auditing constitutes a viable supervised workflow for LLM-assisted contract development.

At the same time, the evidence collected here is equally clear about what current technology cannot yet deliver: production-ready smart contracts generated without expert oversight, evaluation frameworks that replace manual review, or visualization tools whose comprehension benefits extend unconditionally to all user populations. These are not failures of the research program; they are an honest empirical characterization of a frontier that is actively advancing.

The blockchain ecosystem continues to grow in scope and adoption. The Total Value Locked in DeFi protocols, the proliferation of DAO-governed organizations, and the growing interest of legal institutions in on-chain contract enforcement all point toward a future in which the demand for smart contract automation will extend well beyond the population of experts in Solidity development [7]. The research directions identified in this thesis, such as domain-specific fine-tuning, intermediate representation-guided generation, semantic correctness verification, and broad-population visualization, constitute the technical program through which that future can be made accessible, secure, and equitable. The work reported here provides an empirical foundation and a set of reusable tools, such as open-source pipelines, evaluation frameworks, and a validated comprehension tool, on which that program can be built.

References

[1] Emanuele Antonio Napoli, Noemi Romani, Valentina Gatteschi, and Claudio Schifanella. Light and shadows of smart contract development with LLMs. *Expert Systems with Applications*, 296:129011, 2025.

[2] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. -, 2008.

[3] Nick Szabo. Smart Contracts. http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html.

[4] Nick Szabo. The idea of smart contracts, 1997.

[5] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. Technical report, Ethereum Project. Yellow Paper. https://ethereum.github.io/yellowpaper/paper.pdf.

[6] Fabian Schär. Decentralized finance: On blockchain- and smart contract-based financial markets. *Federal Reserve Bank of St. Louis Review*, 103(2):153–174, 2021.

[7] Sam Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William Knottenbelt. SoK: Decentralized finance (DeFi). In *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, pages 30–46, 2022.

[8] Qin Wang, Rujia Li, Qi Wang, and Shiping Chen. Non-Fungible Token (NFT): Overview, Evaluation, Opportunities and Challenges. arXiv preprint arXiv:2105.07447, 2021. https://arxiv.org/abs/2105.07447.

[9] Taewon Ko, Jungho Lee, and Daeyoung Ryu. Survey on Blockchain-Based Non-Fungible Tokens: History, Technologies, Standards, and Open Challenges. *International Journal of Network Management*, 34(1):e2245, 2024.

[10] Carlos Santana and Laura Albareda. Blockchain and the Emergence of Decentralized Autonomous Organizations (DAOs): An Integrative Model and Research Agenda. *Technological Forecasting and Social Change*, 182:121806, 2022.

[11] Samer Hassan and Primavera De Filippi. Decentralized Autonomous Organization. *Internet Policy Review*, 10(2):1–10, 2021.

[12] Nir Kshetri. Blockchain and Sustainable Supply Chain Management in Developing Countries. *International Journal of Information Management*, 60:102376, 2021.

[13] Shuai Wang, Liwei Ouyang, Yong Yuan, Xiaochun Ni, Xuan Han, and Fei-Yue Wang. Blockchain-Enabled Smart Contracts: Architecture, Applications, and Future Trends. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 49(11):2266–2277, 2019.

[14] Reza Soltani, Uyen Trang Nguyen, and Aijun An. A Survey of Self-Sovereign Identity Ecosystem. *Security and Communication Networks*, 2021:8873429, 2021.

[15] Alexander Mühle, Andreas Grüner, Tatiana Gayvoronskaya, and Christoph Meinel. A Survey on Essential Components of a Self-Sovereign Identity. *Computer Science Review*, 30:80–86, 2018.

[16] Vikram Dhillon, David Metcalf, and Max Hooper. The DAO hacked. In *Blockchain Enabled Applications*, pages 67–78. Apress, 2017.

[17] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Comput. Surv.*, 53(3), jun 2020.

[18] William George Bodell, III, Siddharth Gupta, and Yilong Shi. Proxy pattern and its implications on smart contract upgradeability. In *Proceedings of the IEEE International Conference on Blockchain (Blockchain 2023)*. IEEE, 2023.

[19] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15. IEEE, May 2019.

[20] ConsenSys. Mythril: Security analysis tool for evm bytecode.

[21] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019.

[22] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 254–269, New York, NY, USA, 2016. Association for Computing Machinery. https://eprint.iacr.org/2016/633.

[23] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*, pages 9–16, 2018.

[24] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 67–82, 2018.

[25] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 15(3):1–45, 2024.

[26] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8), December 2024.

[27] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR)*, 2024.

[28] Gavina Baralla, Simona Ibba, Lodovica Marchesi, Michele Marchesi, and Elia Riccardi. Let's Write a Smart Contract! Assessment of the GitHub Copilot AI for Smart Contract Development. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE.

[29] Gavina Baralla, Giacomo Ibba, and Roberto Tonelli. Assessing github copilot in solidity development: Capabilities, testing, and bug fixing. *IEEE Access*, 2024.

[30] Alessandro Barbara and Emanuele Napoli. Automatic smart contract generation through LLMs: When the stochastic parrot fails. *Workshop on Distributed Ledger Technologies (DLT 2024)*, 2024.

[31] Felix Härer and Hans-Georg Fill. A comparison of approaches for visualizing blockchains and smart contracts. *Jusletter IT*, February 2019. 21 February 2019, ISSN 1664-848X.

[32] Natkamon Tovanich, Nicolas Heulot, Jean-Daniel Fekete, and Petra Isenberg. Visualization of blockchain data: A systematic review. *IEEE Transactions on Visualization and Computer Graphics*, 27(7):3135–3152, July 2021.

[33] Sandro Curty, Felix Härer, and Hans-Georg Fill. Design of blockchain-based applications using model-driven engineering and low-code/no-code platforms: A structured literature review. *Software and Systems Modeling*, 2023.

[34] Emanuele Antonio Napoli, Ivan Spada, Noemi Romani, Valentina Gatteschi, Claudio Schifanella, et al. Smart contract visualization: Solutions and challenges. In *Proceedings of the Seventh Distributed Ledger Technology Workshop (DLT 2025)*. CEUR, 2025.

[35] João F. Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. Smartbugs: A framework to analyze solidity smart contracts. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1349–1352, Sep. 2020.

[36] Emanuele Antonio Napoli and Valentina Gatteschi. Evaluating chatgpt for smart contracts vulnerability correction. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1828–1833. IEEE, 2023.

[37] Emanuele Antonio Napoli, Federico Barbàra, Valentina Gatteschi, and Claudio Schifanella. Leveraging large language models for automatic smart contract generation. In *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 701–710. IEEE, IEEE, 2024.

[38] GovTech Data Science and Singapore Government AI Division. Prompt engineering playbook.

[39] Massimo Bartoletti and Livio Pompianu. An empirical analysis of smart contracts: Platforms, applications, and design patterns. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *Financial Cryptography and Data Security*, pages 494–509, Cham, 2017. Springer International Publishing.

[40] Faizan Khan, Istvan David, Daniel Varro, and Shane McIntosh. Code cloning in smart contracts on the ethereum platform: An extended replication study. *IEEE Transactions on Software Engineering*, 49(4):2006–2019, 2023.

[41] Emanuele Antonio Napoli, Lorenzo Gangemi, Silvio Meneguzzo, Noemi Romani, and Valentina Gatteschi. Making Smart Contracts Easier To Understand. In *Proceedings of the IEEE Conference*. IEEE. https://github.com/GangemiLorenzo/MasterDegreeThesis.

[42] Sandra G. Hart and Lowell E. Staveland. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In P. A. Hancock and N. Meshkati, editors, *Human Mental Workload*, pages 139–183. North-Holland, 1988.

[43] Leo Maxim Bach, Branko Mihaljevic, and Mario Zagar. Comparative analysis of blockchain consensus algorithms. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1545–1550. IEEE, May 2018.

[44] Muhammad Nasir Mumtaz Bhutta, Amir A. Khwaja, Adnan Nadeem, Hafiz Farooq Ahmad, Muhammad Khurram Khan, Moataz A. Hanif, Houbing Song, Majed Alshamari, and Yue Cao. A survey on blockchain technology: Evolution, architecture and security. *IEEE Access*, 9:61048–61073, 2021.

[45] Rafael Belchior, André Vasconcelos, Sérgio Guerreiro, and Miguel Correia. A survey on blockchain interoperability: Past, present, and future trends. *ACM Computing Surveys*, 54(8):1–41, 2021.

[46] Sara Saberi, Mahtab Kouhizadeh, Joseph Sarkis, and Lejia Shen. Blockchain technology and its relationships to sustainable supply chain management. *International Journal of Production Research*, 57(7):2117–2135, 2019.

[47] Bela Gipp, Norman Meuschke, and André Gernandt. Decentralized trusted timestamping using the crypto currency Bitcoin. In *Proceedings of the iConference 2015*, Newport Beach, CA, USA, 2015.

[48] Zhengping Liu, Hailing He, Lieping Zhang, and Chen Peng. 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, page 255–259, New York, NY, USA, 2024. Association for Computing Machinery.

[49] Jianning Li, Amin Dada, Behrus Puladi, Jens Kleesiek, and Jan Egger. Chatgpt in healthcare: A taxonomy and systematic review. *Computer Methods and Programs in Biomedicine*, 245, 2024. Cited by: 0; All Open Access, Green Open Access, Hybrid Gold Open Access.

[50] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. MedRec: Using blockchain for medical data access and permission management. In *Proceedings of the 2nd International Conference on Open and Big Data (OBD 2016)*, pages 25–30. IEEE, 2016.

[51] Wei Cai, Zehua Wang, Jason B. Ernst, Zhen Hong, Chen Feng, and Victor C. M. Leung. Decentralized applications: The blockchain-empowered software system. *IEEE Access*, 6:53019–53033, 2018.

[52] Anna Kobusinska and Grzegorz Wilczynski. Blocked-based solidity - a service for graphically creating the smart contracts in solidity programming language. In *55th Hawaii International Conference on System Sciences, HICSS 2022, Virtual Event / Maui, Hawaii, USA, January 4-7, 2022*, pages 1–10. ScholarSpace, 2022.

[53] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. Technical report, Ethereum Foundation, 2014. https://ethereum.org/en/whitepaper/.

[54] A.M. Antonopoulos and G. Wood. *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media, Incorporated, 2018.

[55] Xiwei Xu, Ingo Weber, Mark Staples, Liming Zhu, Jan Bosch, Len Bass, Cesare Pautasso, and Paul Rimba. A taxonomy of blockchain-based systems for architecture design. In *Proceedings of the IEEE International Conference on Software Architecture (ICSA 2018)*, pages 243–252. IEEE, 2018.

[56] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts sok. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, volume 10204 of *Lecture Notes in Computer Science*, page 164–186, Berlin, Heidelberg, 2017. Springer-Verlag.

[57] Noama Fatima Samreen and Manar H. Alalfi. Reentrancy vulnerability identification in ethereum smart contracts. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 22–29, 2020.

[58] Hongjun Wu, Zhuo Zhang, Shangwen Wang, Yan Lei, Bo Lin, Yihao Qin, Haoyu Zhang, and Xiaoguang Mao. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In Zhi Jin, Xuandong Li, Jianwen Xiang, Leonardo Mariani, Ting Liu, Xiao Yu, and Nahgmeh Ivaki, editors, *32nd IEEE International Symposium on Software Reliability Engineering, ISSRE 2021, Wuhan, China, October 25-28, 2021*, pages 378–389. IEEE, 2021.

[59] Lucian A. Trestioreanu, Wazen M. Shbair, Flaviene Scheidt de Cristo, and Radu State. Blockly2Hooks: Smart Contracts for Everyone with the XRP Ledger and Google Blockly. In *2023 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, pages 145–150, July 2023. ISSN: 2835-3498.

[60] T. Tateishi, S. Yoshihama, N. Sato, and S. Saito. Automatic smart contract generation using controlled natural language and template. *IBM Journal of Research and Development*, 63(2/3):6:1–6:12, March 2019.

[61] Xuling Ye and Markus König. From the graphical representation to the smart contract language: a use case in the construction industry. In Chen Feng, Thomas Linner, Ioannis Brilakis, Daniel Castro, Po-Han Chen, Yong Cho, Jing Du, Semiha Ergan, Borja Garcia de Soto, Jozef Gašparík, Firas Habbal, Amin Hammad, Kepa Iturralde, Thomas Bock, Soonwook Kwon, Zoubeir Lafhaj, Nan Li, Ci-Jyun Liang, Bharadwaj Mantha, Ming Shan Ng, Daniel Hall, Mi Pan, Wei Pan, Farzad Rahimian, Benny Raphael, Anoop Sattineni, Christian Schlette, Isaac Shabtai, Xuesong Shen, Pingbo Tang, Jochen Teizer, Yelda Turkan, Enrique Valero, and Zhenhua Zhu, editors, *Proceedings of*

the 38th International Symposium on Automation and Robotics in Construction (ISARC), pages 272–279, Dubai, UAE, November 2021. International Association for Automation and Robotics in Construction (IAARC).

[62] Luca Guida and Florian Daniel. Supporting reuse of smart contracts through service orientation and assisted development. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pages 59–68, 2019.

[63] Maximilian Wöhrer and Uwe Zdun. Domain specific language for smart contract development. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9. IEEE, May 2020.

[64] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3(null):1137–1155, March 2003.

[65] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[66] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[67] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS 2017)*, volume 30, pages 5998–6008, 2017.

[68] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT 2019)*, volume 1, pages 4171–4186. Association for Computational Linguistics, 2019.

[69] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS 2020)*, volume 33, pages 1877–1901, 2020.

[70] Hugo Touvron, Thibaut Lavril, Gautier Izacard, et al. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[71] Hugo Touvron, Louis Martin, Kevin Stone, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[72] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[73] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[74] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35, 2023.

[75] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. A systematic survey of prompt engineering in large language models: Techniques and applications. *arXiv preprint arXiv:2402.07927*, 2024.

[76] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS 2022)*, volume 35, pages 24824–24837, 2022.

[77] Lenz Belzner, Thomas Gabor, and Martin Wirsing. Large language model assisted software engineering: Prospects, challenges, and a case study. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 14380 LNCS:355 – 374, 2024. Cited by: 2.

[78] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 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)*, pages 31–53, 2023.

[79] Fangfang Zhou, Yilun Fan, Shenglan Lv, Lijia Jiang, Zhuo Chen, Jian Yuan, Feijiang Han, Haojin Jiang, Genghuai Bai, and Ying Zhao. Fctree: Visualization of function calls in execution. *Inf. Softw. Technol.*, 175(C), November 2024.

[80] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. In *Proceedings of the 2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 23–30. IEEE.

[81] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[82] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. Smart Contract: Attacks and Protections. *IEEE Access*, 8:24416–24427, 2020.

[83] Rick Fonteijn. Comparison of Static Analysis Tooling for Smart Contracts on the EVM. Technical report, University of Twente, 2018. 28th Twente Student Conference on IT.

[84] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, pages 530–541. ACM, 2020.

[85] Julian Aron Prenner, Hlib Babii, and Romain Robbes. Can OpenAI's Codex Fix Bugs? An Evaluation on QuixBugs. In *Proceedings of the Third IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 69–75. ACM, 2022.

[86] Matthew Jin, Syed Shahriar, Shuyao Shi, Wenliang Shen, Haifeng Tang, Jiulong Lu, and Shi Han. InferFix: End-to-End Program Repair with LLMs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1646–1656. ACM.

[87] OpenAI. New embedding models and API updates. https://openai.com/blog/new-embedding-models-and-api-updates, 2024. Documents the `gpt-4-0125-preview` checkpoint.

[88] Niall Roche, Walter Hernández, Eason Chen, Jérôme Siméon, and Dan Selman. Ergo – a programming language for smart legal contracts. In *CoRR*, volume abs/2112.07064, 2021.

[89] Eason Chen, Niall Roche, Yuen-Hsien Tseng, Walter Hernandez, Jiangbo Shangguan, and Alastair Moore. Conversion of legal smart contracts using NLP. In *Companion Proceedings of the ACM Web Conference 2023 (WWW'23 Companion)*, pages 1112–1118, New York, NY, USA, 2023. Association for Computing Machinery.

[90] Pengcheng Fang, Zhenhua Zou, Xusheng Xiao, and Zhuotao Liu. 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, page 727–739, New York, NY, USA, 2023. Association for Computing Machinery.

[91] Emanuel Regnath and Sebastian Steinhorst. Smaconat: Smart contracts in natural language. In *2018 Forum on Specification and Design Languages (FDL)*, pages 5–16, Sep. 2018.

[92] Olivia Choudhury, Nolan Rudolph, Issa Sylla, Noor Fairoza, and Amar Das. Auto-generation of smart contracts from domain-specific ontologies and semantic rules. In *2018 IEEE International Conference on Internet of Things*

(iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), pages 963–970, 2018.

[93] Mohammad Hamdaqa, Lucas Alberto Pineda Met, and Ilham Qasse. iContractML 2.0: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms. *Information and Software Technology*, 144:106762, 2022.

[94] E Chen, Bohan Qin, Yan Zhu, Weijing Song, Shengdian Wang, Cheng-Chung William Chu, and Stephen S. Yau. Spesc-translator: Towards automatically smart legal contract conversion for blockchain-based auction services. *IEEE Transactions on Services Computing*, 15(5):3061–3076, Sep. 2022.

[95] Zhihao Hao, Bo Zhang, Dianhui Mao, Jerome Yen, Zhihua Zhao, Min Zuo, Haisheng Li, and Cheng-Zhong Xu. A novel method using LSTM-RNN to generate smart contracts code templates for improved usability. *Multimedia Tools and Applications*, 82(27):41669–41699, 2023.

[96] Ilham A. Qasse, Shailesh Mishra, Björn Thór Jónsson, Foutse Khomh, and Mohammad Hamdaqa. Chat2code: A chatbot for model specification and code generation, the case of smart contracts. In *2023 IEEE International Conference on Software Services Engineering (SSE)*, pages 50–60, July 2023.

[97] Nenad Petrović and Issam Al-Azzoni. Model-driven smart contract generation leveraging ChatGPT. In *Advances in Systems Engineering (ICSEng 2023)*, volume 761 of *Lecture Notes in Networks and Systems*, pages 387–396. Springer, Cham, 2023.

[98] Giacomo Ibba, Marco Ortu, Roberto Tonelli, and Giuseppe Destefanis. Leveraging ChatGPT for automated smart contract repair: A preliminary exploration of GPT-3-based approaches. 2023. Available at SSRN 4474678.

[99] Roberto Tonelli, Giuseppe Antonio Pierro, Marco Ortu, and Giuseppe Destefanis. Smart contracts software metrics: A first study. *PLoS ONE*, 18(4):e0281043, 2023.

[100] Mantas Jurgelaitis, Lina Čeponienė, Karolis Butkus, Rita Butkienė, and Vaidotas Drungilas. Mda-based approach for blockchain smart contract development. *Applied Sciences*, 13(1), 2023.

[101] Julius Köpke, Giovanni Meroni, and Mattia Salnitri. Designing secure business processes for blockchains with secbpmn2bc. *Future Generation Computer Systems*, 141:382–398, 2023.

[102] Dianhui Mao, Fan Wang, Yalei Wang, and Zhihao Hao. Visual and User-Defined Smart Contract Designing System Based on Automatic Coding. *IEEE Access*, 7:73131–73143, 2019. Conference Name: IEEE Access.

[103] Kritagya Upadhyay, Ram Dantu, Yanyan He, Abiola Salau, and Syed Badruddoja. 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)*, pages 261–268, 2021.

[104] Karra Greenberg, Evgenia Jane Kitaevich, Siddharth Chaudhari, and Anna Kirkland. Analyzing contracts: State of the field, mixed-methods guiding steps, and an illustrative example. *Law & Social Inquiry*, 49(1):423–450, 2024.

[105] Shriyaa Balaji, Ram Dantu, Kritagya Upadhyay, and Thomas McCullough. 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)*, pages 128–137, 2024.

[106] Tristan Coignion, Clément Quinton, and Romain Rouvoy. 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, page 79–89, New York, NY, USA, 2024. Association for Computing Machinery.

[107] Matthew Renze. The effect of sampling temperature on problem solving in large language models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 7346–7356, Miami, Florida, USA, November 2024. Association for Computational Linguistics.

[108] Xiaodong Gu, Meng Chen, Yalan Lin, Yuhan Hu, Hongyu Zhang, Chengcheng Wan, Zhao Wei, Yong Xu, and Juhong Wang. On the effectiveness of large language models in domain-specific code generation. *ACM Trans. Softw. Eng. Methodol.*, 34(3), February 2025.

[109] Hao Ding, Yizhou Liu, Xuefeng Piao, Huihui Song, and Zhenzhou Ji. Smartguard: An llm-enhanced framework for smart contract vulnerability detection. *Expert Systems with Applications*, 269:126479, 2025.

[110] Xiuliang Duan, Dating Tan, Liangda Fang, Yuyu Zhou, Chaobo He, Ziliang Chen, Lusheng Wu, Guanliang Chen, Zhiguo Gong, Weiqi Luo, and Quanlong Guan. 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, page 6959–6968, New York, NY, USA, 2024. Association for Computing Machinery.

[111] Alina Fastowski and Gjergji Kasneci. Understanding knowledge drift in llms through misinformation. In Marco Piangerelli, Bardh Prenkaj, Ylenia Rotalinti, Ananya Joshi, and Giovanni Stilo, editors, *Discovering Drift Phenomena in Evolving Landscapes*, pages 74–85, Cham, 2025. Springer Nature Switzerland.

[112] Alexander S Choi, Syeda Sabrina Akter, JP Singh, and Antonios Anastasopoulos. The llm effect: Are humans truly using llms, or are they being influenced by them instead? *arXiv preprint arXiv:2410.04699*, 2024.

[113] Duarte Pereira, Fernanda Barbosa, and Carmen Morgado. NextBlocks: An Interactive Block Programming Platform. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2024, pages 590–596, New York, NY, USA, July 2024. Association for Computing Machinery.

[114] S. Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer Berlin Heidelberg, 2007.

[115] Noptanit Chotisarn, Leonel Merino, Xu Zheng, Supaporn Lonapalawong, Tianye Zhang, Mingliang Xu, and Wei Chen. A systematic literature review of modern software visualization. *Journal of Visualization*, 23(4):539–558, August 2020.

[116] Devamardeep Hayatpur, Daniel Wigdor, and Haijun Xia. Crosscode: Multilevel visualization of program execution. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, New York, NY, USA, 2023. Association for Computing Machinery.

[117] Katherine E. Isaacs and Todd Gamblin. Preserving command line workflow for a package management system using ascii dag visualization. *IEEE Transactions on Visualization and Computer Graphics*, 25(9):2804–2820, 2019.

[118] Sabin Devkota and Katherine E. Isaacs. CFGExplorer: Designing a Visual Control Flow Analytics System around Basic Program Analysis Operations. *Computer Graphics Forum*, 37(3):453–464, 2018. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13433.

[119] YoungSeok Yoon, Brad A. Myers, and Sebon Koo. Visualization of fine-grained code change history. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*, pages 119–126, 2013.

[120] Katherine E. Isaacs, Peer-Timo Bremer, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, and Bernd Hamann. Combing the communication hairball: Visualizing parallel execution traces using logical time. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2349–2358, 2014.

[121] Ke Xu, Yun Wang, Leni Yang, Yifang Wang, Bo Qiao, Si Qin, Yong Xu, Haidong Zhang, and Huamin Qu. Clouddet: Interactive visual analysis of anomalous performances in cloud computing systems. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):1107–1117, 2020.

[122] Yifan Sun, Yixuan Zhang, Ali Mosallaei, Michael D. Shah, Cody Dunne, and David Kaeli. Daisen: A Framework for Visualizing Detailed GPU

Execution. *Computer Graphics Forum*, 40(3):239–250, 2021. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14303.

[123] Sverrir Thorgeirsson, Chengyu Zhang, Theo B. Weidmann, Karl-Heinz Weidmann, and Zhendong Su. An Electroencephalography Study on Cognitive Load in Visual and Textual Programming. In *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1*, pages 280–292, Melbourne VIC Australia, August 2024. ACM.

[124] Sverrir Thorgeirsson, Theo B. Weidmann, Karl-Heinz Weidmann, and Zhendong Su. Comparing cognitive load among undergraduate students programming in python and the visual language algot. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2024, page 1328–1334, New York, NY, USA, 2024. Association for Computing Machinery.

[125] Massimo Bartoletti, Lorenzo Benetollo, Michele Bugliesi, Silvia Crafa, Giacomo Dal Sasso, Roberto Pettinau, Andrea Pinna, Mattia Piras, Sabina Rossi, Stefano Salis, Alvise Spanò, Viacheslav Tkachenko, Roberto Tonelli, and Roberto Zunino. Smart contract languages: A comparative analysis. *Future Generation Computer Systems*, 164:107563, 2025.

[126] Santiago Bragagnolo, Henrique Rocha, Marcus Denker, and Stephane Ducasse. Smartinspect: solidity smart contract inspector. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 9–18, 2018.

[127] Tim Weingaertner, Rahul Rao, Jasmin Ettlin, Patrick Suter, and Philipp Dublanc. Smart Contracts Using Blockly: Representing a Purchase Agreement Using a Graphical Programming Language. In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 55–64, June 2018.

[128] Michele Marchesi, Lodovica Marchesi, and Roberto Tonelli. An agile software engineering method to design blockchain applications. In *Proceedings of the 14th Central and Eastern European Software Engineering Conference Russia*, CEE-SECR '18, New York, NY, USA, 2018. Association for Computing Machinery.

[129] Péter Garamvölgyi, Imre Kocsis, Benjámin Gehl, and Attila Klenik. Towards model-driven engineering of smart contracts for cyber-physical systems. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 134–139, 2018.

[130] Orlenys López-Pintado, Luciano García-Bañuelos, Marlon Dumas, Ingo Weber, and Alexander Ponomarev. Caterpillar: A business process execution engine on the Ethereum blockchain. *Software: Practice and Experience*, 49(7):1162–1193, 2019. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2702.

[131] Maryam Ghaffari Saadat, Reiko Heckel, and Fernando Orejas. Unfolding symbolic attributed graph grammars. In Fabio Gadducci and Timo Kehrer, editors, *Graph Transformation*, pages 75–90, Cham, 2020. Springer International Publishing.

[132] Sean Tan, Sourav S Bhowmick, Huey Eng Chua, and Xiaokui Xiao. LATTE: Visual Construction of Smart Contracts. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 2713–2716, New York, NY, USA, May 2020. Association for Computing Machinery.

[133] Marek Skotnica and Robert Pergl. Das contract - a visual domain specific language for modeling blockchain smart contracts. In David Aveiro, Giancarlo Guizzardi, and José Borbinha, editors, *Advances in Enterprise Engineering XIII*, pages 149–166, Cham, 2020. Springer International Publishing.

[134] Giuseppe Antonio Pierro. Smart-graph: Graphical representations for smart contract on the ethereum blockchain. In *28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021*, pages 708–714. IEEE, 2021.

[135] Marina Luiza Lardizabal Vieira and Patricia Vilain. Representation of smart contracts as state diagrams. In *2021 IEEE/ACS 18th International Conference on Computer Systems and Applications (AICCSA)*, pages 1–8, 2021.

[136] Marek Skotnica, Jan Klicpera, and Robert Pergl. Towards model-driven smart contract systems-code generation and improving expressivity of smart contract modeling. *Proc. EEWC*, 20, 2020.

[137] Jie Meng, Zheng Li, Ruiliang Zhao, and Ying Shang. An Automated Modeling Method and Visualization Implementation of Smart Contracts. In Hong-Ning Dai, Xuanzhe Liu, Daniel Xiapu Luo, Jiang Xiao, and Xiangping Chen, editors, *Blockchain and Trustworthy Systems*, pages 399–406, Singapore, 2021. Springer.

[138] Mpyana Mwamba Merlec, Youn Kyu Lee, and Hoh Peter In. SmartBuilder: A Block-based Visual Programming Framework for Smart Contract Development. In *2021 IEEE International Conference on Blockchain (Blockchain)*, pages 90–94, December 2021.

[139] Kevin Purnell and Rolf Schwitter. User-defined smart contracts using answer set programming. In Guodong Long, Xinghuo Yu, and Sen Wang, editors, *AI 2021: Advances in Artificial Intelligence*, pages 291–303, Cham, 2022. Springer International Publishing.

[140] Farouk Hdhili, Ramy Gouiaa, and Marc Jansen. Implementation and evaluation of a visual programming language in the context of blockchain. In Javier Prieto, Alberto Partida, Paulo Leitão, and António Pinto, editors, *Blockchain and Applications*, pages 74–82, Cham, 2022. Springer International Publishing.

[141] Reiko Heckel, Zobia Erum, Nitia Rahmi, and Albert Pul. Visual smart contracts for daml. In Nicolas Behr and Daniel Strüber, editors, *Graph Transformation*, pages 137–154, Cham, 2022. Springer International Publishing.

[142] Stefano Bistarelli, Francesco Faloci, and Paolo Mori. Towards a graphical dsl for tracing supply chains on blockchain. In Ricardo Chaves, Dora B. Heras, Aleksandar Ilic, Didem Unat, Rosa M. Badia, Andrea Bracciali, Patrick Diehl, Anshu Dubey, Oh Sangyoon, Stephen L. Scott, and Laura Ricci, editors, *Euro-Par 2021: Parallel Processing Workshops*, pages 219–229, Cham, 2022. Springer International Publishing.

[143] Jesús Rosa-Bilbao, Juan Boubeta-Puig, and Adrian Rutle. Edaloco: Enhancing the accessibility of blockchains through a low-code approach to the development of event-driven applications for smart contract management. *Computer Standards and Interfaces*, 84:103676, 08 2023.

[144] Mojtaba Eshghie, Wolfgang Ahrendt, Cyrille Artho, Thomas Troels Hildebrandt, and Gerardo Schneider. Capturing smart contract design with dcr graphs. In Carla Ferreira and Tim A. C. Willemse, editors, *Software Engineering and Formal Methods*, pages 106–125, Cham, 2023. Springer Nature Switzerland.

[145] Peter Bodorik, Christian Gang Liu, and Dawn Jutla. TABS: Transforming automatically BPMN models into blockchain smart contracts. *Blockchain: Research and Applications*, 4(1):100115, March 2023.

[146] Flavio Corradini, Alessandro Marcelletti, Andrea Morichetta, Andrea Polini, Barbara Re, and Francesco Tiezzi. A flexible approach to multi-party business process execution on blockchain. *Future Generation Computer Systems*, 147:219–234, 2023.

[147] Xinzhe Shen, Wenyu Li, Hanchuan Xu, Xiao Wang, and Zhongjie Wang. A reuse-oriented visual smart contract code generator for efficient development of complex multi-party interaction scenarios. *Applied Sciences*, 13(14):8094, 2023.

[148] Peng Qin, Quanyi Hu, and Menglin Cui. Towards machine-readable semantic-based e-business contract representations using network of timed automata (nta). *Future Generation Computer Systems*, 158:457–471, 2024.

[149] Xuling Ye, Ningshuang Zeng, Xingyu Tao, Daguang Han, and Markus König. Smart contract generation and visualization for construction business process collaboration and automation: Upgraded workflow engine. *Journal of Computing in Civil Engineering*, 38(6):04024030, 2024.

[150] Xiaolin Wen, Tai D. Nguyen, Shaolun Ruan, Qiaomu Shen, Jun Sun, Feida Zhu, and Yong Wang. Ponzilens+: Visualizing bytecode actions for smart ponzi scheme identification. *IEEE Transactions on Visualization and Computer Graphics*, pages 1–14, 2024.

[151] Yun-Cheng Tsai, Jiun-Yu Huang, and Da-Ru Chiou. Empowering young learners to explore blockchain with user-friendly tools: a method using Google Blockly and NFTs. *IET Blockchain*, 4(4):324–334, 2024. _eprint: https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/blc2.12055.

[152] Cristian Gómez-Macías, Juan Manuel Vara, Francisco Javier Pérez-Blanco, and David Granada. A block-based web ide to ease the smart contract programming learning curve. *IEEE Revista Iberoamericana de Tecnologias del Aprendizaje*, 19:321–330, 01 2024.

[153] Jun Jin, Le Yan, Yidan Zou, Jie Li, and Zhen Yu. Research on smart contract verification and generation method based on bpmn. *Mathematics*, 12(14):2158, 2024.

[154] Aicha El Abidi and Hamid Elghazi. Converting bpmn diagrams into solidity code: A comprehensive new approach and algorithm. In *2024 IEEE International Conference on Technology Management, Operations and Decisions (ICTMOD)*, pages 1–6. IEEE, 2024.

[155] Xinzhe Shen, Zhongjie Wang, Jiale Luo, Haoqi Ruan, Hanchuan Xu, and Mingyi Liu. Ibc: An integrated framework combining blockchain with bpmn choreography to enhance multi-party collaboration. In *2024 IEEE International Conference on Web Services (ICWS)*, pages 457–467. IEEE, 2024.

[156] Aicha El Abidi, Hamid El Ghazi, and Saïd Assar. An integration model for the mapping of bpmn to smart contract: Mcq case study. *IEEE Engineering Management Review*, 2024.

[157] Simon Curty and Hans-Georg Fill. Smartcml: A visual modeling language to enhance the comprehensibility of smart contract implementations. In Elda Paja, Jelena Zdravkovic, Evangelia Kavakli, and Janis Stirna, editors, *The Practice of Enterprise Modeling*, pages 87–104, Cham, 2025. Springer Nature Switzerland.

[158] Ali Samanipour, Omid Bushehrian, and Gregorio Robles. Mdapw3: Mda-based development of blockchain-enabled decentralized applications. *Science of Computer Programming*, 239:103185, 2025.

[159] Sheng Gao, Wenting Liu, Jianming Zhu, Xuewen Dong, and Jin Dong. Bpmn-llm: Transforming bpmn models into smart contracts using large language models. *IEEE Software*, 2025.

[160] Hao Luo, Yuhao Lin, Xiao Yan, Xintong Hu, Yuxiang Wang, Qiming Zeng, Hao Wang, and Jiawei Jiang. Guiding LLM-based smart contract generation with finite state machine. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence (IJCAI-25)*, 2025.

[161] Jeffy Yu et al. Retrieval augmented generation integrated large language models in smart contract vulnerability detection. *arXiv preprint arXiv:2407.14838*, 2024.

[162] Ye Liu et al. PropertyGPT: LLM-driven formal verification of smart contracts through retrieval-augmented property generation. In *arXiv preprint arXiv:2405.02580*, 2024.

Appendix A

Used Prompts

The seventeen CO-STAR prompt templates used in Chapters 4, 5, and 6 are reproduced below. Each template instantiates a Context persona, an Objective, an optional Requirements block, an optional Style, and an optional Audience; the placeholder <legal agreement> marks the position at which the source legal document is inserted.

PR1

> You are a Masters student in Computer Science. Write a smart contract in Solidity based on the following legal agreement
>
> <legal agreement>

PR2

> You are a Masters student in Computer Science. 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 your professor.

PR3

> You are a Masters student in Computer Science. 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 your professor.

PR4

You are a Masters student in Computer Science. 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 your professor.

PR5

> You are a Masters student in Computer Science. 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 professor.

PR6

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

PR7

> You are a junior 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 your supervisor.

PR8

> You are a junior 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 your supervisor.

PR9

You are a junior 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 your supervisor.

PR10

You are a junior 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 your supervisor.

PR11

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

<legal agreement>

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.