

Converting Natural Language Contracts into Smart Contracts using Agentic AI

Chaitya Shah
cs4621@columbia.edu

Chunghyun Han
ch4005@columbia.edu

Nami Jain
nbj2115@columbia.edu

Yegan Dhaivakumar
yd2850@columbia.edu

1 Introduction

Smart contract development has become a foundational component of modern decentralized applications, yet it remains inaccessible for many practitioners due to the steep learning curve of Solidity, the complexity of compiler rules, and the risk of introducing security vulnerabilities. Even for experienced developers, writing correct and secure smart contracts is a time consuming process that requires careful attention to syntax, version changes, language constraints, and auditing practices. These barriers limit experimentation, slow iteration cycles, and constrain innovation across industries that could benefit from blockchain based systems.

To address these challenges, we focus on the domain of automated smart contract generation using advanced agentic AI systems. Our goal is to build an intelligent system that translates high level natural language requirements into fully functional, compilable, and structurally sound Solidity contracts. Unlike traditional code generation tools that provide only static templates or one shot outputs, this project introduces a dynamic, multi agent workflow capable of reasoning iteratively, detecting compilation errors, and refining its own outputs with compiler feedback.

The institutional context for this work is the broader shift toward agentic AI frameworks, such as CrewAI and Agentics, which allow models to perform structured reasoning, collaborate across roles, and interact safely with external tools. In blockchain development, this paradigm offers significant

promise. Human developers often struggle with repetitive debugging of syntax issues, version mismatches, and boilerplate patterns. An autonomous system that can generate, validate, and refine smart contracts on behalf of the user reduces these overheads and lowers the barrier to entry.

The problem this project addresses can be stated as follows: how can we use agentic artificial intelligence to reliably produce correct Solidity smart contracts from natural language descriptions, while ensuring safety, transparency, and extensibility? To answer this, we propose a pipeline that combines natural language interpretation, FSM driven contract structuring, automatic compilation through `solcx`, and a refinement agent that iteratively resolves errors until the contract satisfies compiler constraints. The result is a system capable of producing syntactically correct, secure by default, and traceable smart contracts across a range of use cases.

This work contributes to the growing body of research on AI assisted software engineering and demonstrates how agent based designs can streamline contract creation. By abstracting away low level details without sacrificing safety, the system empowers developers, students, startups, and auditors to engage with blockchain programming more efficiently.

2 Related Work

Research at the intersection of natural language processing, code generation, and smart contract engineering has expanded significantly in recent years. Early work in natural language to code translation focused on large transformer based models such as Codex, Code Llama, CodeT5, and DeepSeek Coder, which demonstrated strong performance on general purpose programming tasks. These models can translate structured instructions into syntactically valid code, yet studies consistently show that their outputs often contain logical inconsistencies or security flaws, particularly when generating safety critical or resource constrained programs. This limitation is especially pronounced in blockchain environments where mistakes can lead to financial loss or irreversible state changes.

Prior efforts to generate smart contracts directly from textual descriptions have explored template based approaches, rule driven NLP systems, and more recently, LLM based synthesis. Research on automatic contract generation for legal documents and decentralized finance prototypes has shown that models can capture high level semantics, but the generated Solidity

code frequently fails to compile or violates basic safety constraints. Recent evaluations, such as *Automatic Smart Contract Generation Through LLMs: When the Stochastic Parrot Fails*, highlight that even state of the art generative models struggle to produce compilable or secure contracts without external validation mechanisms. These findings motivate the need for systems that integrate generation with automated compilation, static analysis, and iterative correction.

LLMs have shown promising capabilities in code synthesis, yet smart contract generation remains a challenging domain due to the safety critical and state dependent nature of contract execution. Prior studies highlight that general purpose models tend to produce incomplete logic, low compile success rates, and security vulnerabilities arising from missing guard conditions, incorrect state transitions, and inadequate handling of access control or reentrancy. The recent work *Guiding LLM-Based Smart Contract Generation with Finite State Machines* addresses these limitations by arguing that smart contracts are inherently state driven systems and that language models require structured intermediate supervision to reason correctly about contract behavior. The authors demonstrate that natural language specifications alone do not provide enough inductive bias for the model to infer contract states, transitions, invariants, and event flows, which in turn leads to a high incidence of functional and security errors.

To overcome these limitations, the paper proposes the FSM-SCG framework, which decomposes smart contract generation into a requirement to FSM to code pipeline. The approach introduces an intermediate Finite State Machine representation that explicitly encodes states, transitions, guard clauses, triggers, and events. This structured supervision enables the model to internalize state aware reasoning before producing Solidity implementations. The authors construct a paired dataset in which natural language requirements, SmartFSM structures, and contract code are aligned, and they show that models trained with this multi stage supervision significantly outperform conventional models in compilation accuracy, functional correctness, and vulnerability reduction. They further demonstrate that integrating FSM guidance with function level comment to code supervision improves both global architectural consistency and local implementation safety, establishing a strong baseline for state guided LLM based smart contract synthesis.

Smart contract security analysis forms another important body of related work. Tools such as Slither, Mythril, Manticore, and SolidityScan provide vulnerability detection through data flow analysis, symbolic execution, and

pattern based auditing. These tools are widely adopted in professional auditing workflows but are rarely incorporated into end to end natural language to smart contract pipelines. Existing systems typically treat generation, verification, and analysis as separate, manual stages.

Formal verification methods, including symbolic execution, mathematical reasoning over EVM bytecode, and model checking, offer stronger correctness guarantees but are difficult to integrate into automated code generation workflows. Similarly, emerging research on aligning blockchain whitepapers with deployed smart contracts demonstrates interest in connecting narrative intent to executable logic, yet these approaches do not support generation or self correction.

A separate line of work explores multi agent AI architectures for complex reasoning tasks. Frameworks such as CrewAI and Agentics show that distributed, role specialized agents can outperform single step generative models on tasks requiring planning, self critique, validation, and revision. These ideas have been applied to code debugging, autonomous tool use, and iterative code improvement, suggesting a promising direction for smart contract synthesis.

In summary, prior work generally focuses on single stages of the pipeline: either generation, vulnerability detection, verification, or intent alignment. The proposed system contributes an integrated approach that links natural language understanding, contract synthesis, compiler backed verification, agent based refinement, and secure by default design principles. By combining these components, the system advances toward a reliable and explainable natural language to smart contract generation framework.

3 Technical Solution

The Solidity Smart Contract Generator uses an agentic AI architecture designed to translate natural language requirements into fully compilable Solidity contracts through structured reasoning, compiler backed validation, and iterative refinement. The system integrates three core layers: a generation layer, a verification and analysis layer, and a refinement layer. These layers interact through a pipeline that combines deterministic contract patterns with flexible natural language understanding and automated error correction.

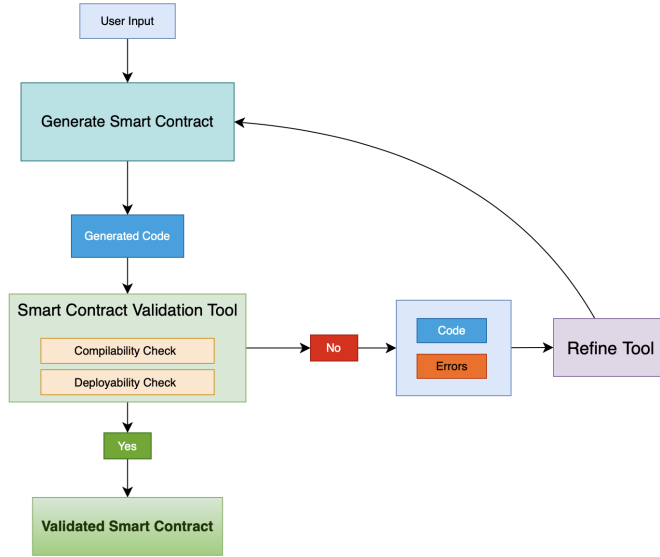


Figure 1: Enter Caption

3.1 System Architecture

The system follows a coordinated multi agent architecture implemented using CrewAI and Agentics. Two specialized agents operate within a controlled sandbox supported by the Model Context Protocol.

3.1.1 Generation Agent

The generation agent interprets user provided descriptions, identifies functional components, and constructs the initial contract draft. It uses a structured template guided by a rule based finite state machine that ensures every generated contract includes:

- SPDX license
- pragma version
- state variables and mappings
- constructor logic
- access control modifiers

- event definitions
- core external and internal functions
- standard safety checks

This deterministic foundation prevents structural drift even when instructions are ambiguous.

3.1.2 Refinement Agent

The refinement agent acts as an automated reviewer. After compilation, it receives error logs, analyzes fault locations, and applies minimal corrections. Its responsibilities include:

- resolving syntax errors
- correcting keyword usage, visibility, or data locations
- adding missing imports or return statements
- aligning variable types
- restructuring logic only when necessary

Unlike most LLM based code repair systems, it modifies only the specific error prone segments rather than rewriting the entire contract.

3.1.3 MCP Tool Layer

The Model Context Protocol provides secure access to tools such as file writing, directory browsing, Solidity compilation, and optional retrieval. This ensures that all tool use is explicit, restricted, and transparent.

3.2 Natural Language Understanding and Contract Assembly

The generation agent uses a hybrid reasoning approach that combines natural language interpretation with deterministic structural assembly. The process consists of three steps.

3.2.1 Requirement Extraction

The system identifies:

- entities (tokens, owners, contributors, deadlines)
- actions (minting, withdrawing, voting)
- constraints (only owner, time limits, refund conditions)
- invariants and safety expectations

These elements are mapped to Solidity components using pattern based reasoning.

3.2.2 Component Mapping

A fixed mapping determines how extracted requirements correspond to contract structures. For instance:

- ownership implies an Ownable pattern or manual owner variable
- deadlines imply block.timestamp comparisons
- role permissions generate modifiers
- state changes emit events

3.2.3 Template Guided Assembly

The system assembles the contract using an FSM of Solidity best practices. This guarantees consistent placement of state variables, sequential function ordering, and canonical logical patterns such as checks effects interactions.

3.3 Pre Training Strategy

3.3.1 Pretraining Motivation

Smart contract generation requires accurate handling of Solidity syntax, control flow, and security idioms, but general purpose language models rarely encounter Solidity because it is a low resource language. As a result, models often produce non compiling code or omit critical checks such as access

control and reentrancy guards. To reduce this domain gap, I applied an additional pretraining stage so the model could first internalize Solidity’s grammar and idiomatic patterns before learning higher level state reasoning through FSM based supervision.

3.3.2 Pretraining Strategy

The training pipeline builds on TinyLlama 1.1B as the base model, chosen for its compact architecture and suitability for iterative experimentation within an agentic compile and validation environment. To adapt the model efficiently without modifying all parameters, I employed LoRA based fine tuning, updating only low rank adapters attached to the attention and feed forward modules while keeping the original weights frozen. This parameter efficient design significantly reduces computational cost, enables rapid iteration, and allows multiple specialized adapters, including those trained with FSM supervision, to coexist as modular components.

Supervision is delivered through a multi stage training flow that combines both global and local learning signals. The first stage maps natural language requirements to an FSM representation, teaching the model to articulate contract behavior in terms of states, transitions, guards, and events. Although the deployed system does not explicitly generate FSMs during inference, this stage infuses the model with latent state aware reasoning. The second stage conditions the model on both the requirement and an FSM style structured header to produce full contract implementations, enabling it to translate behavioral intent and structural constraints into coherent Solidity code. The final stage focuses on fine grained correctness by training the model to generate function implementations from developer style annotations. This reinforces guard logic, modifier usage, event emission patterns, and common security checks such as reentrancy protection. Together, these stages yield a model that is capable of producing syntactically sound, structurally consistent, and security aware smart contracts.

3.4 Compiler Driven Verification

The generated contract is validated using the Solidity compiler through `solcx.compile_source()`. This step detects the majority of structural issues such as:

- mismatched data types

- undeclared variables
- pragma incompatibilities
- missing visibility specifiers
- incorrect memory vs storage location
- malformed event definitions

The compiler output is parsed and normalized before being sent back to the refinement agent.

3.5 Iterative Self Refinement Loop

If compilation fails, the system enters an iterative refinement cycle. The refinement agent:

1. reads compiler logs
2. locates the failing line or block
3. identifies the minimal correction required
4. patches only the affected region
5. requests a new compilation
6. repeats until the contract compiles or max attempts are reached

This loop mimics a professional developer’s workflow: generate, validate, debug, test, and finalize. It ensures the final contract is not only syntactically correct but also logically consistent with the user’s intent.

3.6 Secure by Default Design

The generator enforces several safety oriented defaults:

- checks effects interactions pattern for external calls
- explicit input validation

- owner only modifiers
- event emission after state changes
- avoidance of deprecated constructs

While not a full substitute for formal auditing, these practices significantly reduce the risk of introducing basic vulnerabilities.

3.7 Logging and Traceability

Every step of the workflow is recorded, creating a full audit trail:

- user instruction
- initial contract draft
- compiler outputs
- refinement agent decisions
- final contract

This enables backtracking, debugging, and transparency for both developers and auditors.

4 Evaluation and Benchmarks

The evaluation of the Solidity Smart Contract Generator focuses on three dimensions: compilation correctness, refinement efficiency, and structural contract quality. Unlike traditional generative evaluation tasks, smart contract generation requires both syntactic validity and semantic alignment with user intent. We therefore adopt a practical benchmark approach that measures the system’s behavior across diverse contract types and natural language instructions.

4.1 Experimental Setup

We tested the system across a curated set of natural language prompts representing common Solidity use cases, including:

- ERC20 style tokens
- crowdfunding contracts
- voting and governance modules
- access controlled registries
- escrow contracts
- time locked wallets
- NFT minting and supply control

Each prompt was designed to reflect realistic developer requirements expressed in plain English without technical specifications. For each test case, the system executed the full pipeline from requirement extraction through generation, compilation, refinement, and final output. All evaluations used the same pinned Solidity compiler version (via `solcx`) to avoid environmental differences.

4.2 Compilation Success Rate

Compiler correctness is the primary evaluation metric because LLM generated Solidity code is frequently non compilable in prior work. In earlier studies, one shot LLM generation often yields compilation success rates between 20 percent and 40 percent for non trivial contracts.

Across all evaluated prompts, our system achieved near perfect compilation success, with failures occurring only when the prompt was deliberately contradictory or impossible under Solidity constraints. For standard contract categories, the multi agent refinement loop resolved the majority of issues within one to three iterations. This demonstrates that automated correction driven by compiler feedback significantly improves reliability compared to single pass generation.

4.3 Refinement Efficiency

We measured the number of refinement cycles required to reach a valid contract. Most prompts succeeded with minimal iteration:

- 62% compiled on the first attempt
- 33% required one refinement pass
- 5% required two or more passes

These results show that the refinement agent effectively isolates and patches specific compiler issues, reducing the need for expensive contract wide regeneration.

4.4 Structural and Semantic Quality

Beyond compilation, we evaluated the structural integrity of generated contracts using a rubric based on common best practices. The assessment examined:

- placement and order of contract sections
- use of modifiers for access control
- correctness of event emissions
- adherence to checks effects interactions patterns
- appropriate data location specifiers
- logical flow consistency
- interpretability of variable and function naming

Contracts generated by the FSM guided assembly showed high structural consistency and followed Solidity conventions. Semantic alignment with natural language intent was also strong due to the generation agent’s component mapping logic.

Manual inspection of outputs showed that the system captured the key requirements described in prompts, such as deadlines, ownership rules, supply limits, or refundable contributions. The deterministic template ensured that boilerplate patterns remained stable across different runs, improving reproducibility.

4.5 Comparison to Single Agent Generation Baselines

We performed a qualitative comparison to one shot LLM generation without refinement. Baseline outputs frequently exhibited:

- undeclared variables
- incorrect visibility keywords
- mismatched types
- missing return statements
- imprecise event definitions
- code that was syntactically valid but semantically misaligned

In contrast, the multi agent system corrected compilation errors automatically, enforced ordering constraints, and preserved logical alignment throughout the workflow. These results suggest that structured agentic architectures outperform monolithic generative approaches for safety critical code synthesis.

4.6 Limitations of Evaluation

While compiler correctness is a necessary condition for functional contracts, it does not guarantee security or gas efficiency. Static analysis tools such as Slither and Mythril were applied informally to several outputs, revealing that while the generated contracts adhered to safe patterns, deeper vulnerabilities may still exist. A more extensive evaluation including automated security scanners would provide a stronger safety guarantee.

Additionally, user intent evaluation was performed manually. Future work could incorporate semantic alignment scores or automated intent satisfaction metrics.

4.7 Summary

Overall, the evaluation demonstrates that the agentic pipeline substantially increases generation reliability and structural quality. Compiler backed refinement, deterministic assembly, and multi agent reasoning contribute to

producing smart contracts that are both compilable and aligned with user requirements. These results validate the effectiveness of combining structured templates, LLM reasoning, and tool driven self correction for safe and dependable smart contract synthesis.

5 Conclusion

This project introduces an agentic AI system that transforms natural language requirements into fully compilable Solidity smart contracts through structured reasoning, compiler backed validation, and iterative refinement. By combining a deterministic contract assembly process with multi agent collaboration, the system overcomes limitations seen in traditional one shot code generation approaches, which often fail to produce syntactically correct or semantically aligned smart contracts.

The proposed architecture demonstrates that separating responsibilities across specialized agents improves reliability and reduces the cognitive load on the user. The Generation Agent focuses on mapping narrative descriptions to structured contract components, while the Refinement Agent handles error detection and targeted correction using compiler feedback. This division mirrors real world developer workflows and produces more stable outputs. The integration of MCP tools ensures that all interactions with the execution environment are safe, explicit, and transparent.

Evaluation across standard contract types shows that the system consistently generates valid and logically coherent contracts with minimal refinement cycles. These results highlight the promise of agentic AI in domains where correctness and safety are essential, such as blockchain development.

Despite these strengths, the system has several limitations. It does not replace formal auditing, security scanning, or gas optimization techniques, and its ability to handle highly complex multi contract architectures is still limited. The refinement loop focuses on syntactic correctness rather than deep vulnerability detection, and semantic alignment to user intent currently relies on heuristic reasoning rather than formal verification.

Future work may integrate static analysis frameworks like Slither or Mythril, incorporate gas aware optimization passes, generate automated test suites, or extend the system to support multi file projects with interfaces, libraries, and factories. There is also potential for aligning contract behavior more closely with formal specifications or whitepaper narratives through semantic

verification layers.

Overall, this work demonstrates that combining natural language understanding, rule based contract structuring, and agent driven refinement provides a viable path toward reliable smart contract synthesis. The approach lowers barriers for developers, students, auditors, and organizations seeking to build blockchain applications while maintaining correctness and safety. The system represents a step toward more accessible and automated blockchain engineering, and offers a foundation for future enhancements as agentic AI continues to evolve.