Guiding LLM-based Smart Contract Generation with Finite State Machine

Hao Luo[1,†] , **Yuhao Lin**[1,†] , **Xiao Yan**[1] , **Xintong Hu**[2] , **Yuxiang Wang**[1] , **Qiming Zeng**[1] , **Hao Wang**[1,*] and **Jiawei Jiang**[1,*]
[1]School of Computer Science, Wuhan University
[2]School of Cyber Science and Engineering, Wuhan University
{lohozz, yuhao_lin}@whu.edu.cn, yanxiaosunny@gmail.com, {xintong.hu, nai.yxwang, kirin_z, wanghao.cs, jiawei.jiang}@whu.edu.cn

Abstract

Smart contract is a kind of self-executing code based on blockchain technology with a wide range of application scenarios, but the traditional generation method relies on manual coding and expert auditing, which has a high threshold and low efficiency. Although Large Language Models (LLMs) show great potential in programming tasks, they still face challenges in smart contract generation w.r.t. effectiveness and security. To solve these problems, we propose FSM-SCG, a smart contract generation framework based on finite state machine (FSM) and LLMs, which significantly improves the quality of the generated code by abstracting user requirements to generate FSM, guiding LLMs to generate smart contracts, and iteratively optimizing the code with the feedback of compilation and security checks. The experimental results show that FSM-SCG significantly improves the quality of smart contract generation. Compared to the best baseline, FSM-SCG improves the compilation success rate of generated smart contract code by at most 48%, and reduces the average vulnerability risk score by approximately 68%.

1 Introduction

Blockchain, with its characteristics of immutability, transparency, and decentralization, has demonstrated a wide range of applications in various fields such as finance, supply chain, data sharing, etc [Wang *et al.*, 2018]. As a key component for realizing blockchain systems, the automatic generation of smart contracts has become one of the focuses in current research [Khan *et al.*, 2021].

Smart contracts require precise coding and verification to ensure their effectiveness and security [Almakhour *et al.*, 2020]. Due to the irreversible nature of smart contract execution, any vulnerabilities or errors in the code could lead to serious security risks, or even financial losses [Luu *et al.*, 2016; Sayeed *et al.*, 2020]. Therefore, the ability to quickly and reliably generate fully functional and secure smart contracts has become a critical issue. Previously, the generation of smart contracts primarily relies on manual coding and professional audits of experts, requiring developers to have extensive blockchain programming skills and experience [Mao *et al.*, 2019].

Table 1: Comparing the existing smart contract generation methods. FSM-SCG stands for our proposed approach, while FSM refers to the variant that only uses the proposed SmartFSM as an intermediate representation to aid smart contract generation. Higher CPR indicates better effectiveness, and lower VRS means better security. We evaluate the methods on the LlaMa3.1-8b model.

Method	Pattern	CPR(↑)	VRS(↓)
Direct	R2C	36.9%	7.44
CML	R2M2C	47.5%	6.36
IContractML	R2M2C	46.3%	6.21
FSM (Ours)	R2F2C	49.3%	6.05
FSM-SCG (Ours)	R2F2C	**95.1%**	**2.36**

In recent years, LLMs have demonstrated significant potential in programming tasks, such as code generation, program repair, and code translation [Poesia *et al.*, 2022; Jiang *et al.*, 2024b]. Motivated as such, LLMs have also shifted the paradigm of smart contract generation. Existing works on LLMs for generating smart contracts mainly focus on direct generation from user requirements to code (R2C) [Napoli *et al.*, 2024; Chatterjee and Ramamurthy, 2024]. Other works transform requirements into intermediate models to guide LLMs in code generation (requirements-to-model-to-code, R2M2C) [Wöhrer and Zdun, 2020; Petrović and Al-Azzoni, 2023; Qasse *et al.*, 2023].

As shown in Table 1, when the LLMs directly generate smart contracts based on user requirements, the code has a low compile pass rate (CPR) and a high vulnerability risk score (VRS). When intermediate states, such as the Contract Modeling Language (CML), IContractML, and the FSM in our proposed method, are introduced to guide the LLMs to generate smart contracts, all metrics significantly improve.

Challenges. Despite pioneering efforts to generate smart contracts using LLMs, there are notable limitations concerning their essential features.

- Effectiveness: Smart contracts typically use low-resource programming languages, such as Solidity, Vyper, and Move, which lack sufficient documentation, libraries, and community support. Consequently, in the internal repre-

[†]Equal Contribution
[*]Corresponding Author

sentation space of LLMs, the distribution of low-resource programming languages is often more concentrated compared to high-resource programming languages, such as Python or JavaScript [Li *et al.*, 2024]. This disparity leads to a higher occurrence of syntax errors and incomplete logic in generated smart contracts, resulting in low compilation success rates.

- Security: Existing research [Chatterjee and Ramamurthy, 2024; Jiang *et al.*, 2024a] has found that LLMs frequently overlook security issues when generating smart contract code. This oversight stems from the use of general-purpose LLM services (e.g., ChatGPT, OpenAI Codex), which are not specifically designed for the blockchain domain, potentially introducing security vulnerabilities in the generated contracts.

To address these limitations, we try to study the following question in this work — *How to leverage LLMs to generate effective and secure smart contracts?*

Our solution FSM-SCG. Formal methods enable precise modeling of requirements and systems to ensure high reliability and security. FSM, as a formal method, can capture the state changes and condition triggers in smart contract to guide secure and reliable generation[Suvorov and Ulyantsev, 2019]. Therefore, we consider incorporating FSM into the smart contract generation process. Additionally, inspired by the Chain-of-Thought (CoT)[Wei *et al.*, 2022] and Structured Chain-of-Thought (SCoT)[Li ⇳ *et al.*, 2023] techniques, we divide the smart contract generation process into two subtasks: generating FSMs from user requirements and generating smart contracts from FSMs. Our approach follows a requirement-to-FSM-to-code (R2F2C) generation pattern, guided by FSM throughout the process, which we call FSM-SCG. The workflow begins by fine-tuning LLM to enhance its ability to generate FSMs from user requirements, and then generate smart contracts from FSMs. Then, we format user requirements, with which the fine-tuned LLMs generate FSMs. After this, the generated FSMs undergo format and graph checks. Once verified, FSMs are input into LLMs to generate the corresponding smart contract code. The resulting contracts are evaluated in terms of effectiveness and security. These validations are utilized as feedback to refine the generation results.

To summarize, we make the following contributions.

- We propose a smart contract generation framework called FSM-SCG upon LLMs. By abstracting user requirements as FSM, FSM-SCG can significantly improve the generation of smart contracts.
- We construct a fine-tuning dataset to improve the ability of LLMs to generate FSMs and smart contracts. Additionally, we conduct compilation and security checks, and use them as feedback to optimize the generation of smart contracts.
- Experimental results show that FSM-SCG significantly enhances the effectiveness and security of smart contracts. Using LlaMa3.1-8B, the compilation success rate reaches 95.1%, 48% higher than the best baseline. Security risks are greatly reduced, with the vulnerability risk score dropping by 68% on average.

2 Preliminaries

2.1 Smart Contract Generation Process

There are two approaches to generating smart contracts using LLMs. The first is the direct generation approach, which inputs user requirements directly for the LLM to generate corresponding smart contract code. The second is the intermediate representation approach, where the LLM first transforms user requirements into an abstract model, then generates smart contract code from this model, effectively decomposing the task into two steps. The direct generation approach prioritizes rapid implementation and user interaction, while the intermediate representation approach focuses on modeling and abstraction, making it better suited for complex requirements. Our work adopts the second approach, utilizing FSM as the intermediate representation.

2.2 Finite State Machine

FSM is an abstract computational model used to represents the dynamic behavior of a system by recognizing input sequences and transitioning between states according to predefined rules.

A Mealy state machine is a type of FSM. Its main characteristic is that the output depends not only on the current state but also on the input signal, allowing for an instantaneous and precise response when the input changes. Typically, a Mealy state machine can be defined as a quintuple $(S, X, Y, \delta, \lambda)$:

- S represents the set of states;
- X represents the set of inputs;
- Y represents the set of outputs;
- $\delta : S \times X \to S$ is the state transition function, which defines the transition rules under different inputs;
- $\lambda : S \times X \to Y$ is the output function that determines the system output based on the current state and input.

The Mealy state machine, although it can represent the state transfer of smart contracts well, has limited guidance for the contract generation task.

3 FSM Guided Smart Contract Generation

In this part, we first provide an overview of our solution FSM-SCG. Then, we introduce its key designs, including an enhanced FSM to encode more information for smart contracts, the prompts tailored for generating FSM and smart contract, and the refinement of smart contract using feedback from compilation and security checks.

3.1 FSM-SCG Overview

Figure 1 provides an overview of FSM-SCG, including the following key processes. We also provide the detailed algorithm of FSM-SCG in Appendix A.

Fine-tuning Process. To enhance the ability of LLMs in generating smart contracts, we build a fine-tuning dataset D_{ft} that uses FSM as an intermediate representation to generate smart contracts based on user requirements (see Section 3.2). Based on this dataset, we optimize the pretrained LLM M_{pre} using the supervised full parameter fine-tuning (FPFT) method [Kenton and Toutanova, 2019]. In FPFT, all

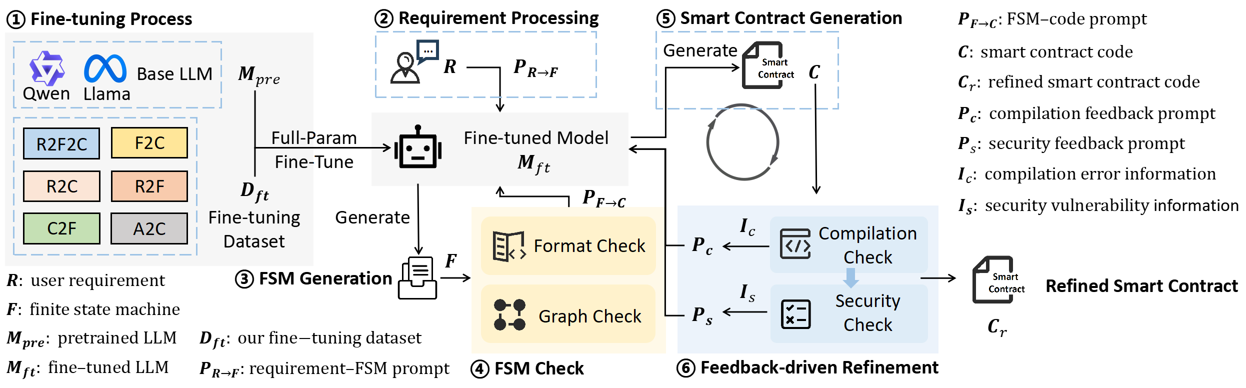

① Fine-tuning Process
Qwen
Llama
Base LLM
M_{pre}
R2F2C
F2C
R2C
R2F
C2F
A2C
Full-Param Fine-Tune
D_{ft}
Fine-tuning Dataset
② Requirement Processing
R
$P_{R\to F}$
Fine-tuned Model M_{ft}
Generate
③ FSM Generation
F
$P_{F\to C}$
Format Check
Graph Check
④ FSM Check
⑤ Smart Contract Generation
Generate
Smart Contract
C
I_c
P_c
Compilation Check
I_s
P_s
Security Check
⑥ Feedback-driven Refinement
Refined Smart Contract
C_r
R: user requirement
F: finite state machine
M_{pre}: pretrained LLM
M_{ft}: fine-tuned LLM
D_{ft}: our fine-tuning dataset
$P_{R\to F}$: requirement-FSM prompt
$P_{F\to C}$: FSM-code prompt
C: smart contract code
C_r: refined smart contract code
P_c: compilation feedback prompt
P_s: security feedback prompt
I_c: compilation error information
I_s: security vulnerability information

Figure 1: An overview of our FSM-SCG framework.

model parameters are updated during training, enabling the model to learn the task-specific domain. In this manner, the LLM can better understand user requirements, translate them into FSMs, and master the domain-specific syntax and semantics of smart contracts. Together, these adaptations significantly improve the quality of the generated code.

Requirement processing. This module takes the natural language descriptions of the desired smart contract from the user, where the user requirements are denoted as R. We transform R into the Requirement-FSM Prompt $P_{R\to F}$, which can guide the LLM to generate the FSM F describing the desired smart contract.

FSM generation. The Requirement-FSM Prompt $P_{R\to F}$ is fed into the fine-tuned LLM M_{ft}. The M_{ft} analyzes the functionality requirements and logic within the user requirements, extracts the states and their transition conditions, and generates the corresponding FSM F.

FSM check. This module conducts format and graph checks on FSM F. The format check ensures syntactic and content integrity, while the graph check verifies logical correctness of states and transitions. Finally, the FSM-Code Prompt $P_{F\to C}$ is generated to produce the smart contract C.

Smart contract generation. The checked FSM F and FSM-Code Prompt $P_{F\to C}$ are fed into fine-tuned LLM M_{ft} to guide the generation of smart contract C. Since F is rigorously checked in terms of format and logic, M_{ft} can efficiently encapsulate its states and events in smart contracts.

Feedback Driven Refinement. Generated smart contracts C are evaluated for effectiveness and security. We use Py-solc-x to check for compilation errors I_c and Slither [Feist *et al.*, 2019] to detect vulnerabilities I_s. These issues are fed back into the LLM via compilation P_c and security P_s prompts to guide refinement, producing refined smart contracts C_r.

3.2 Fine-tune Dataset Construction

Existing work has not provided open-source datasets [Liu *et al.*, 2024; Zhao *et al.*, 2024; Luo *et al.*, 2024], making it difficult to reproduce their results. Moreover, these works consider a single task or scenario and thus cannot encompass the diverse requirements of practical smart contract generation. To tackle these two problems, we construct an open-source fine-tuning dataset that covers various tasks and adopts a dialogic format.

We collect smart contract source code from platforms like Etherscan and use GPT-4o to generate the corresponding user requirements and FSM, forming a dataset of 30k items, each containing requirements (R), FSM (F), and code (C). From this, we derive sub-datasets by rearranging elements—*R2F2C*, *R2F*, *F2C*, *C2F*, and *R2C*. Additionally, the *A2C* dataset maps extracted annotations to function code. These datasets support training LLMs for various smart contract generation tasks. We present the dataset details in Appendix B.

3.3 Enhanced FSM for Smart Contract

Limitations of Mealy. As in Section 2.2, the Mealy machine abstracts state transitions and outputs but lacks explicit representation of functions, variables, and state-output interactions, limiting its ability to capture smart contract complexity.

SmartFSM. To enhance the capability of FSM to represent smart contracts, we propose *SmartFSM*, an enhancement to the Mealy state machine that more accurately models the structure and functionality of smart contracts. SmartFSM divides the representation of contracts into five sections, i.e., *basic information*, *states*, *variables*, *functions*, and *events*, which we introduce as follows.

- Basic information. This section contains the background and functional descriptions of the contract, providing an overall overview of the contract for LLM.
- States. This section lists all possible states of the contract to help the LLM understand the states and transfer logic.
- Variables. This section records important data in the smart contract, serving as the conditions for state transfer.
- Functions. Each functional module of a smart contract is usually a function that receives external inputs and may trigger state transfers. The function part resembles the input-output mechanism in Mealy state machine.
- Events. This section records the key operations and state changes that occur in the contract. The triggering of each event reflects the operational state of the contract and is similar to the output function in the Mealy state machine.

Dividing smart contract into five sections, SmartFSM retains Mealy machine state transfer capabilities while better representing variables, functions, and underlying details. We provide an example in Appendix C.

3.4 Checking of SmartFSM

The SmartFSM generated by LLMs may have errors like incomplete structure or incorrect logic. Therefore, we verify its correctness through three steps: SmartFSM information extraction, format check, and graph check.

SmartFSM information extraction. From SmartFSM, we extract the following sets: The state set $S = \{s_1, s_2, \ldots, s_n\}$, representing all possible states. The trigger set $X = \{x_1, x_2, \ldots, x_m\}$, representing all triggers causing state transitions. The target state set $T = \{t_1, t_2, \ldots, t_k\}$, where $T \subseteq S$, representing all possible target states. The transition set $\Delta = \{(s, x, t) \mid s \in S, x \in X, t \in T\}$, representing all transitions as triples.

Format check. This step uses static analysis methods to verify the completeness and correctness of the SmartFSM. The correct FSM should satisfy the following conditions: (i) the initial state s_0 must be defined within the state set ($s_0 \in S$); (ii) the target states in transitions must be defined within the state set ($\forall(s, x, t) \in \Delta,\ t \in S$); (iii) the triggers must be defined within the event set ($\forall(s, x, t) \in \Delta,\ x \in X$).

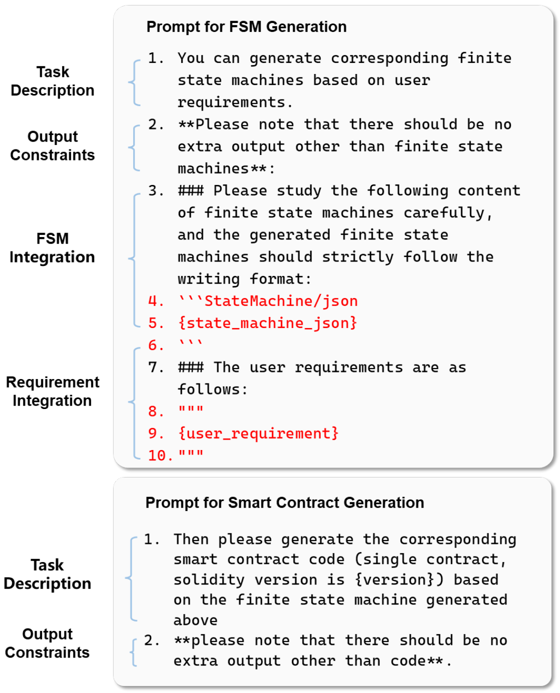
Graph check. Format check ensures that the SmartFSM follows the formatting rules but it does not ensure the SmartFSM is logically correct. Graph check is used to check logical correctness as the states and state transfers of FSM can be expressed as graphs. We first use the extracted states to construct a directed graph, where the nodes represent the states, the edges represent state transitions, and the states are connected through state transitions. A SmartFSM can be considered correct if it satisfies the following conditions: (i) All states in the SmartFSM can be reached from the initial state via some path p ($\forall s \in S, \exists p : s_0 \to s$); (ii) The state graph should have loops but no self-loops, to ensure that states can transition correctly ($t \neq s$).

3.5 Prompts for FSM and Contract Generation

Prompt engineering is crucial for effectively using LLMs. Figure 2 shows the prompts for FSM and contract generation.

Prompt for FSM generation. To construct FSM accurately from user requirements, we carefully design the prompt to make the LLM focus on the structure of the FSM and ensure that no unnecessary information is generated. The prompt comprises the following components:

- Task description: Guide the LLM to generate smart contracts based on user requirements.
- Output constraints: Request the LLM to return only the FSM content, without outputting any other information.
- FSM integration: The prompt explicitly provides the JSON format of the FSM. By providing a predefined format, the prompt helps the LLM understand the expected structure and syntax of FSM. This also reduces the possibility of formatting errors and deviations from the intended structure.
- Requirement integration: The prompt includes placeholders for user requirements.



Prompt for FSM Generation
Task Description
Output Constraints
FSM Integration
Requirement Integration
Prompt for Smart Contract Generation
Task Description
Output Constraints

```
1. You can generate corresponding finite
   state machines based on user
   requirements.
2. **Please note that there should be no
   extra output other than finite state
   machines**:
3. ### Please study the following content
   of finite state machines carefully,
   and the generated finite state
   machines should strictly follow the
   writing format:
4. ```StateMachine/json
5. {state_machine_json}
6. ```
7. ### The user requirements are as
   follows:
8. """
9. {user_requirement}
10. """
```

```
1. Then please generate the corresponding
   smart contract code (single contract,
   solidity version is {version}) based
   on the finite state machine generated
   above
2. **please note that there should be no
   extra output other than code**.
```

Figure 2: The prompts for FSM and contract generation.

The prompt design, by providing a task description, clearly limiting the output, and offering a structured template, enables the LLM to understand the requirements and generate reliable FSMs, while minimizing the possibility of redundant or irrelevant content.

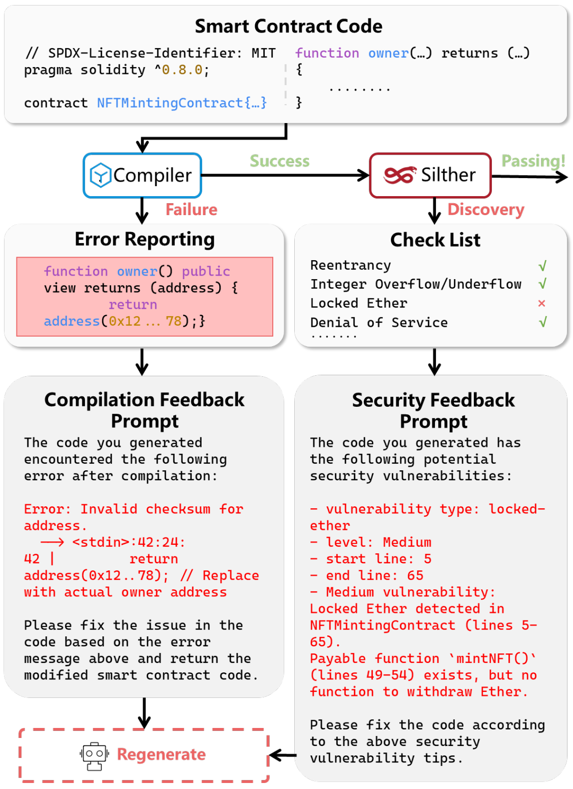
Prompt for smart contract generation. In order to generate a smart contract that meets user requirements, conforms to the specific FSM, and adheres to coding standards, we design the following prompt.

- Task description: As shown at the bottom of Figure 2, the prompt directs the LLM to generate smart contract code based on the previously generated FSM. The LLM can leverage the session context to extract the smart contract content and accurately implement the logic and structure defined in the FSM, without requiring the FSM to be included in the prompt.
- Output constraints: Request the LLM to return the smart contract, without outputting any other information.

The proposed prompt hence maintain logical consistency between the design phase (FSM) and the implementation phase (smart contract). This stratified approach ensures that the behavior represented in the FSM, such as state transitions and triggers, is reflected in the code.

3.6 Contract Refinement with Feedback

Compilation feedback. As shown in Figure 3, we compile the generated code by Solidity compiler. If errors are detected, we return the errors to the LLM along with a contextual prompt to help improve the quality of generated code. In the prompt, we inform the LLM about the details of the errors and desired improvements for regeneration. The feedback not only provides the specific reasons for failure but also



```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract NFTMintingContract{…}
```

```
function owner(…) returns (…)
{
    ........
}
```

```
function owner() public
view returns (address) {
        return
address(0x12 … 78);}
```

```
Reentrancy                    √
Integer Overflow/Underflow    √
Locked Ether                  ×
Denial of Service             √
.......
```

```
The code you generated
encountered the following
error after compilation:

Error: Invalid checksum for
address.
  ⟶ <stdin>:42:24:
42 |         return
address(0x12..78); // Replace
with actual owner address

Please fix the issue in the
code based on the error
message above and return the
modified smart contract code.
```

```
The code you generated has
the following potential
security vulnerabilities:

- vulnerability type: locked-
ether
- level: Medium
- start line: 5
- end line: 65
- Medium vulnerability:
Locked Ether detected in
NFTMintingContract (lines 5-
65).
Payable function `mintNFT()`
(lines 49-54) exists, but no
function to withdraw Ether.

Please fix the code according
to the above security
vulnerability tips.
```

Figure 3: Examples for compilation and security feedback.

offers explicit correction guidance. As Figure 3 shows, during the compilation process, we identify address errors within the functions and provide the errors in the *compilation feedback prompt* to guide the LLM in regenerating the contract.

Security feedback. To ensure code security, we use Slither to detect vulnerabilities such as reentrancy and locked assets. Detected issues, including type and code segment, are recorded in a *security feedback prompt*. Figure 3 shows a "Locked Ether" vulnerability, where the contract can receive but not transfer funds. Using a predefined prompt template, we integrate details like type, level, position, and reason, providing the prompt to the LLM to guide regeneration.

The above feedback procedure is iterated until compilation errors and security vulnerabilities are eliminated.

3.7 Comparison to prior works

Compared to the prior works [Napoli *et al.*, 2024; Chatterjee and Ramamurthy, 2024; Petrović and Al-Azzoni, 2023; Qasse *et al.*, 2023], our approach has the following advantages:

FSM as an Intermediate Representation. Our experiments show that FSM outperforms IContractML[Petrović and Al-Azzoni, 2023] and CML[Wöhrer and Zdun, 2020], as it concisely captures user requirements and state transitions. In contrast, IContractML is overly complex with redundant details, while CML lacks state transition representation.

Fine-tuning and Feedback. We use supervised FPFT approach to enhance smart contract generation and introduce a novel feedback mechanism. Effectiveness and security checks provide feedback to improve contract quality.

4 Experimental Evaluation

4.1 Experiment Settings

Base LLMs. To ensure comprehensiveness, we evaluate our method using both closed-source models (GPT-4o, GeMini1.5-Flash, Qwen-plus) and open-source models (LlaMa3.1-405B/8B, Qwen2.5-7B), covering different parameter scales.

Baselines and Our Method. We select three representative methods for smart contract generation as baselines.

- Direct. A series of research on smart contract generation uses LLMs to generate smart contract code based on user requirements directly [Napoli *et al.*, 2024; Chatterjee and Ramamurthy, 2024].
- IContractML. IContractML first uses the LLMs to generate the IContractML model language, and then generates contract code from the model language [Petrović and Al-Azzoni, 2023; Qasse *et al.*, 2023]. It represents methods that introduce an intermediate representation during smart contract generation.
- CML. Contract Modeling Language (CML), a high-level DSL, has been used in traditional smart contract generation[Wöhrer and Zdun, 2020], but not for guiding LLMs. We adopt CML as an intermediate representation and use it as a baseline for comparison.

We compare the performance of the three variants of the methods proposed in this paper with the baseline.

- FSM. We employ the SmartFSM proposed in Section 3.3 as an intermediate representation to guide LLMs to generate smart contracts. However, it does not incorporate iterative feedback mechanisms or involve fine-tuning the LLM.
- FSM-SCG*. FSM-SCG* extends FSM by incorporating compilation and security feedback to iteratively optimize smart contracts, improving their effectiveness and security without fine-tuning the LLM.
- FSM-SCG. FSM-SCG further enhances FSM-SCG* by fine-tuning the LLM on our fine-tuning dataset.

Environment and Parameters. We fine-tune instruction versions of LlaMa3.1-8B and Qwen2.5-7B models using 8 NVIDIA A6000-48GB GPUs. FPFT is conducted for 3 epochs with the AdamW optimizer, with a learning rate of 5e-5. In FSM-SCG, we perform one round of compilation feedback and one round of security feedback.

4.2 Performance Metrics

CPR. Compilation pass rate (CPR) reflects the effectiveness of the generated smart contracts.

$$CPR = N_{\text{compiled}}/N_{\text{total}} \times 100\%$$

where N_{compiled} is the number of successfully compiled smart contracts, and N_{total} is the total number of contracts.

ZRCP and HRCP. Zero risk contract percentage (ZRCP) and high risk contract percentage (HRCP) quantify the security of contracts.

$$ZRCP = N_{zero}/N_{total}, \qquad HRCP = N_{high}/N_{total}$$

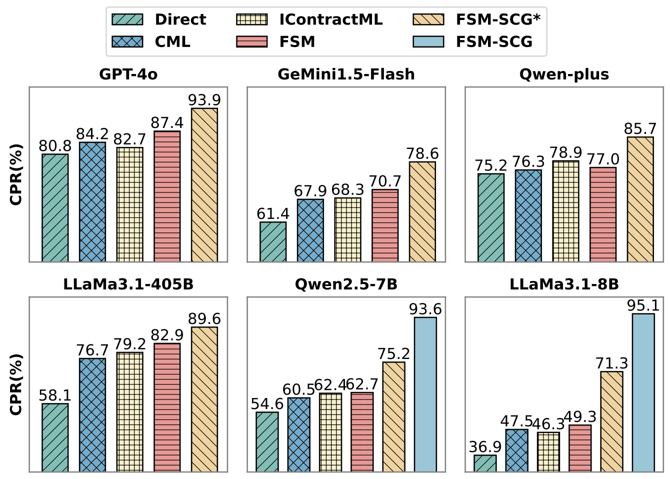


Figure 4: Compilation Pass Rate (CPR) of smart contracts generated by each method on different LLMs. All methods generate contracts from the same 1,000 user requirements.

where N_{zero} is the number of contracts with zero risk scores, N_{high} is the number of contracts with high-severity vulnerabilities, and N_{total} is the total number of contracts.

VRS. Vulnerability risk score (VRS) indicates the risk level, and a high VRS may be caused by many vulnerabilities or high severity of those vulnerabilities.

$$VRS = \frac{\sum_{i=1}^{n} (\text{SeverityScore}_i \times \text{ConfidenceScore}_i)}{n}$$

where n is the total number of detected vulnerabilities, SeverityScore_i is the severity score of vulnerability i, which can be 3 (high), 2 (medium), or 1 (low); and ConfidenceScore_i is the confidence score of vulnerability i, which can be 3 (high), 2 (medium), or 1 (low). VRS is set to 10 when the code does not compile successfully.

4.3 Main Experiments

Effectiveness. We evaluate methods using the CPR of generated contracts. We sample 1,000 high-quality requirements from dataset for testing. For each requirement, LLM generates three code samples, which are compiled to compute CPR. Figure 4 presents the results.

- *Closed-source LLMs.* The FSM method uses SmartFSM as an intermediate representation to guide LLMs, outperforming Direct, CML, and IContractML in most cases on both closed-source and open-source LLMs. The FSM-SCG* method, adding generation feedback, achieves the best results among all baselines.
- *Open-source LLMs.* Due to fine-tuning limitations, FSM-SCG is applied only to LlaMa3.1-8B and Qwen2.5-7B, achieving significant CPR improvements, surpassing even GPT-4o. For example, the CPR of LlaMa3.1-8B rises from 36.9% to 95.3%, proving its effectiveness in enhancing smart contract compilation success.

Our method outperforms baselines by structuring requirements into SmartFSM, covering all states and events to enhance contract generation.

Table 2: Evaluation results for Security. We highlight the best method in **bold** and the second-best with underline.

Model	Approach	ZRCP(↑)	HRCP(↓)	VRS(↓)
GPT-4o	Direct	30.32%	15.47%	3.6720
	CML	35.04%	12.83%	3.4135
	IContractML	34.82%	13.54%	3.4775
	FSM	35.81%	11.90%	3.3235
	FSM-SCG*	**42.75%**	**8.63%**	**2.7649**
GeMini1.5-Flash	Direct	28.90%	36.48%	4.8316
	CML	28.42%	28.72%	4.8043
	IContractML	29.58%	26.94%	4.7821
	FSM	32.69%	23.76%	4.5622
	FSM-SCG*	**36.90%**	**18.58%**	**3.9462**
Qwen-plus	Direct	27.39%	22.74%	4.2730
	CML	35.39%	23.07%	4.2956
	IContractML	37.90%	20.91%	4.2561
	FSM	38.18%	18.12%	4.1063
	FSM-SCG*	**41.77%**	**14.35%**	**3.6983**
LlaMa3.1-405B	Direct	28.22%	15.51%	6.0433
	CML	30.51%	17.47%	4.4756
	IContractML	29.30%	15.28%	4.1419
	FSM	34.39%	16.65%	3.9342
	FSM-SCG*	**39.84%**	**13.62%**	**3.4529**
Qwen2.5-7B	Direct	16.85%	22.43%	7.9704
	CML	19.01%	20.33%	5.7987
	IContractML	20.67%	16.03%	4.9844
	FSM	22.01%	19.62%	5.2419
	FSM-SCG*	30.72%	14.76%	4.0981
	FSM-SCG	**50.53%**	**6.62%**	**2.4832**
LlaMa3.1-8B	Direct	27.10%	21.68%	7.4416
	CML	30.74%	32.21%	6.3598
	IContractML	31.34%	37.80%	6.2111
	FSM	33.27%	20.28%	6.0461
	FSM-SCG*	35.20%	14.87%	4.2540
	FSM-SCG	**53.21%**	**5.15%**	**2.3621**

Security. We evaluate security of each method in contract generation using ZRCP, HRCP, and VRS metrics. We sample 1,000 high-quality requirements from dataset for testing. For each requirement, LLM generates three code samples. Security is assessed through performance testing, as shown in Table 2, with the following observations:

- *Closed-source LLMs.* Although FSM lacks a feedback mechanism, it enhances contract security, outperforming Direct, CML, and IContractML in ZRCP and ranking second in HRCP and VRS. This shows SmartFSM improves secure contract generation, with FSM-SCG* achieving the best overall results.
- *Open-source LLMs.* After applying FSM-SCG, the percentage of smart contracts without security vulnerabilities (ZRCP) increases to 53.21% on Qwen2.5-7B and 50.53% on LlaMa3.1-8B, while high-risk vulnerabilities (HRCP) decrease to 5.15% and 6.62%, respectively.

Our method surpasses baselines in security by ensuring valid state transitions, preventing unsafe changes, handling exceptions, and reducing vulnerabilities.

4.4 Ablation Study

We conduct ablation studies (Table 3) to assess the contributions of key components, including the feedback mechanism, A2C fine-tuning, and the overall fine-tuning process.

Table 3: Ablation studies for fine-tuning and refinement with feedback. We highlight the best method in **bold**.

Model	Approach	CPR(↑)	ZRCP(↑)	HRCP(↓)	VRS(↓)
Qwen2.5-7B	Direct	54.6%	16.85%	22.43%	7.9704
	Mealy FSM	59.8%	19.23%	25.46%	6.2762
	Our FSM	62.7%	22.01%	19.62%	5.2419
	FSM-SCG	**93.6%**	**50.53%**	**6.62%**	**2.4832**
	- w/o Feedback	88.5%	35.71%	14.46%	3.4567
	- w/o A2C Dataset	90.9%	44.00%	8.69%	3.0942
	- w/o Fine-tuning	75.2%	30.72%	14.76%	4.0981
LlaMa3.1-8B	Direct	36.9%	27.10%	21.68%	7.4416
	Mealy FSM	37.3%	29.37%	23.03%	6.6347
	Our FSM	49.3%	33.27%	20.28%	6.0461
	FSM-SCG	**95.1%**	**53.21%**	**5.15%**	**2.3621**
	- w/o Feedback	91.3%	38.12%	11.94%	3.1061
	- w/o A2C Dataset	92.8%	44.40%	8.73%	2.8349
	- w/o Fine-tuning	71.3%	35.20%	14.87%	4.2540

Table 4: The impact of fine-tuning methods on the effect.

Method	Model	CPR(↑)	VRS(↓)
FPFT	Qwen2.5-7B	93.6%	2.48
	LlaMa3.1-8B	95.1%	2.36
LoRA	Qwen2.5-7B	80.5%	3.37
	LlaMa3.1-8B	75.8%	3.56

- **Effect of SmartFSM.** Replacing SmartFSM with Mealy FSM leads to performance degradation. As shown in the table, SmartFSM outperforms Mealy FSM w.r.t. all metrics because it better expresses user requirements and guides LLMs in generating smart contracts.
- **Effect of feedback.** Removing the feedback significantly reduces CPR and worsens security metrics. For example, the CPR of LlaMa3.1-8B drops from 95.1% to 91.3%, and VRS increases from 2.36 to 3.11. This shows that feedback improves the compilation success rate and contract security by iteratively refining the generated contracts.
- **Effect of A2C fine-tuning.** Removing the A2C dataset from our fine-tuning dataset and fine-tuning the LLM causes a performance drop as the A2C dataset improves LLM's ability to associate code with annotation.
- **Effect of entire fine-tuning.** Without fine-tuning, all metrics degrade significantly, highlighting its importance to improve CPR and security. Fine-tuning helps the model grasp Solidity complexities, meet functional requirements, and address security vulnerabilities.
- **Effect of fine-tuning method.** We compare FPFT and LoRA [Hu *et al.*, 2022; Xia *et al.*, 2024] and include the experimental results in Table 4. FPFT outperforms LoRA by updating all parameters for better task adaptation, while LoRA's limited updates hinder adaptability. Thus, we choose FPFT for our experiments. The detailed discussion is in the Appendix D.

4.5 Parameter Sensitivity

We also conduct experiments to evaluate the parameter sensitivity of FSM-SCG (see Appendix D). Specifically, we investigate the impact of feedback counts and fine-tuning epochs.

- **Effect of feedback count.** Varying the feedback count from 0 to 5, the largest improvement occurs from 0 to 1, as most errors are simple and need minimal iterations. Thus, we set the feedback count to 1 for a balance of performance and efficiency.
- **Effect of fine-tuning epochs.** We fine-tune FSM-SCG from 1 to 5 epochs. Performance improves notably up to 3 epochs but plateaus thereafter, as the model converges early and saturates with further training. To balance efficiency and performance, we select 3 epochs.

5 Related Work

Automatic Generation Methods for Smart Contracts. Smart contract generation methods include Model-Driven generation (MD)[López-Pintado *et al.*, 2017; Tran *et al.*, 2018] , Domain-Specific Languages (DSL)[Clack *et al.*, 2019; Skotnica and Pergl, 2019] , and Visual Programming Languages (VPL)[Mavridou and Laszka, 2018; Mao *et al.*, 2019]. MD enhances design-code consistency by converting domain models into smart contracts but requires specialized knowledge. DSL simplifies contract development within specific domains but lacks broader applicability and robust verification. VPL uses graphical interfaces to lower development barriers but offers limited functionality and scalability.

Generation of Smart Contracts Using LLMs. The development of LLMs has made automatic smart contract generation a research focus[Yang *et al.*, 2024]. Chatterjee et al.[Chatterjee and Ramamurthy, 2024] explored using descriptive and structured prompts to generate smart contracts, finding quality issues and neglect of security by most LLMs. Petrović and Al-Azzoni[Petrović and Al-Azzoni, 2023] proposed a model-driven framework using ChatGPT, but it suffers from long response times and high costs. Chen et al.[Yong *et al.*, 2024] improved contract quality by combining AST-LSTM representation, clustering models, prompt dataset optimization, and fine-tuning LLaMA2-7B. Qasse et al.[Qasse *et al.*, 2023] extended DSLs with chatbots, improving functionality but facing challenges with input errors and accuracy. However, LLMs still face issues like incorrect syntax, security vulnerabilities, and limited adaptability in this domain [Huang *et al.*, 2024b; Huang *et al.*, 2024a].

Anatomy of Prior Works. Most prior works, except [Chatterjee and Ramamurthy, 2024], rely on a single LLM for smart contract generation, and none has fine-tuned LLMs for this task. Analyzing a single LLM is insufficient, therefore, our work employs six LLMs within the FSM-SCG framework. We assess contract generation through compilation checks, security analysis, and use case testing. A feedback mechanism further enhances trustworthiness by guiding LLMs to refine output, distinguishing our approach and improving performance [Zhang *et al.*, 2024; Wang *et al.*, 2024].

6 Conclusion

In this work, we propose an FSM-guided framework for smart contract generation with LLMs, improving effectiveness and security. We release a fine-tuning dataset to better align LLMs with user needs and integrate compilation and security checks into a feedback mechanism for improved reliability.

Acknowledgments

This work was sponsored by National Key R&D Program of China (No.2023YFB2703600), Key R&D Program of Hubei Province (2023BAB077), the Fundamental Research Funds for the Central Universities (2042025kf0040), and National Natural Science Foundation of China (62472327, 62441229). This work was supported by Sichuan Clinical Research Center for Imaging Medicine (YXYX2402).

Contribution Statement

Hao Luo and Yuhao Lin contributed equally to this work.

References

[Almakhour *et al.*, 2020] Mouhamad Almakhour, Layth Sliman, Abed Ellatif Samhat, and Abdelhamid Mellouk. Verification of smart contracts: A survey. *Pervasive and Mobile Computing*, 67:101227, 2020.

[Chatterjee and Ramamurthy, 2024] Siddhartha Chatterjee and Bina Ramamurthy. Efficacy of various large language models in generating smart contracts. *arXiv preprint arXiv:2407.11019*, 2024.

[Clack *et al.*, 2019] Christopher D Clack, Vikram A Bakshi, and Lee Braine. Smart contract templates: foundations, design landscape and research directions (2016). *Preprint. arXiv*, 1608, 2019.

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

[Hu *et al.*, 2022] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

[Huang *et al.*, 2024a] Hao Huang, Qian Yan, Keqi Han, Ting Gan, Jiawei Jiang, Quanqing Xu, and Chuanhui Yang. Learning diffusions under uncertainty. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 20430–20437, 2024.

[Huang *et al.*, 2024b] Qiang Huang, Xiao Yan, Xin Wang, Susie Xi Rao, Zhichao Han, Fangcheng Fu, Wentao Zhang, and Jiawei Jiang. Retrofitting temporal graph neural networks with transformer. *arXiv preprint arXiv:2409.05477*, 2024.

[Jiang *et al.*, 2024a] Jiawei Jiang, Hao Huang, Zhigao Zheng, Yi Wei, Fangcheng Fu, Xiaosen Li, and Bin Cui. Detecting and analyzing motifs in large-scale online transaction networks. *IEEE Transactions on Knowledge and Data Engineering*, 2024.

[Jiang *et al.*, 2024b] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

[Kenton and Toutanova, 2019] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*, volume 1, page 2, 2019.

[Khan *et al.*, 2021] Shafaq Naheed Khan, Faiza Loukil, Chirine Ghedira-Guegan, Elhadj Benkhelifa, and Anoud Bani-Hani. Blockchain smart contracts: Applications, challenges, and future trends. *Peer-to-peer Networking and Applications*, 14:2901–2925, 2021.

[Li *et al.*, 2024] Zihao Li, Yucheng Shi, Zirui Liu, Fan Yang, Ali Payani, Ninghao Liu, and Mengnan Du. Quantifying multilingual performance of large language models across languages. *arXiv preprint arXiv:2404.11553*, 2024.

[Liu *et al.*, 2024] Qidong Liu, Xian Wu, Xiangyu Zhao, Yuanshao Zhu, Derong Xu, Feng Tian, and Yefeng Zheng. When moe meets llms: Parameter efficient fine-tuning for multi-task medical applications. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1104–1114, 2024.

[Li ⚨ *et al.*, 2023] Jia Li ⚨ , Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 2023.

[López-Pintado *et al.*, 2017] Orlenys López-Pintado, Luciano García-Bañuelos, Marlon Dumas, and Ingo Weber. Caterpillar: A blockchain-based business process management system. *BPM (Demos)*, 172:1–5, 2017.

[Luo *et al.*, 2024] Ling Luo, Jinzhong Ning, Yingwen Zhao, Zhijun Wang, Zeyuan Ding, Peng Chen, Weiru Fu, Qinyu Han, Guangtao Xu, Yunzhi Qiu, et al. Taiyi: a bilingual fine-tuned large language model for diverse biomedical tasks. *Journal of the American Medical Informatics Association*, page ocae037, 2024.

[Luu *et al.*, 2016] 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*, pages 254–269, 2016.

[Mao *et al.*, 2019] 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.

[Mavridou and Laszka, 2018] Anastasia Mavridou and Aron Laszka. Tool demonstration: Fsolidm for designing secure ethereum smart contracts. In *Principles of Security and Trust: 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software*, pages 270–277, 2018.

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

[Petrović and Al-Azzoni, 2023] Nenad Petrović and Issam Al-Azzoni. Model-driven smart contract generation leveraging chatgpt. In *International Conference On Systems Engineering*, pages 387–396, 2023.

[Poesia *et al.*, 2022] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227*, 2022.

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

[Sayeed *et al.*, 2020] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. Smart contract: Attacks and protections. *IEEE Access*, 8:24416–24427, 2020.

[Skotnica and Pergl, 2019] Marek Skotnica and Robert Pergl. Das contract-a visual domain specific language for modeling blockchain smart contracts. In *Enterprise Engineering Working Conference*, pages 149–166, 2019.

[Suvorov and Ulyantsev, 2019] Dmitrii Suvorov and Vladimir Ulyantsev. Smart contract design meets state machine synthesis: Case studies. *arXiv preprint arXiv:1906.02906*, 2019.

[Tran *et al.*, 2018] An Binh Tran, Qinghua Lu, and Ingo Weber. Lorikeet: A model-driven engineering tool for blockchain-based business process execution and asset management. In *BPM (dissertation/demos/industry)*, pages 56–60, 2018.

[Wang *et al.*, 2018] Shuai Wang, Yong Yuan, Xiao Wang, Juanjuan Li, Rui Qin, and Fei-Yue Wang. An overview of smart contract: architecture, applications, and future trends. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 108–113, 2018.

[Wang *et al.*, 2024] Yuxiang Wang, Xiao Yan, Shiyu Jin, Hao Huang, Quanqing Xu, Qingchen Zhang, Bo Du, and Jiawei Jiang. Self-supervised learning for graph dataset condensation. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 3289–3298, 2024.

[Wei *et al.*, 2022] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

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

[Xia *et al.*, 2024] Yifei Xia, Fangcheng Fu, Wentao Zhang, Jiawei Jiang, and Bin Cui. Efficient multi-task llm quantization and serving for multiple lora adapters. *Advances in Neural Information Processing Systems*, 37:63686–63714, 2024.

[Yang *et al.*, 2024] Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Shaochen Zhong, Bing Yin, and Xia Hu. Harnessing the power of llms in practice: A survey on chatgpt and beyond. *ACM Transactions on Knowledge Discovery from Data*, 18(6):1–32, 2024.

[Yong *et al.*, 2024] Chen Yong, Hu Defeng, Xu Chao, Chen Nannan, Fanfan Shen, and Jianbo Liu. Smart contract generation model based on code annotation and ast-lstm tuning, 2024.

[Zhang *et al.*, 2024] Qinbo Zhang, Xiao Yan, Yukai Ding, Quanqing Xu, Chuang Hu, Xiaokai Zhou, and Jiawei Jiang. Treecss: An efficient framework for vertical federated learning. In *International Conference on Database Systems for Advanced Applications*, pages 425–441. Springer, 2024.

[Zhao *et al.*, 2024] Zihuai Zhao, Wenqi Fan, Jiatong Li, Yunqing Liu, Xiaowei Mei, Yiqi Wang, Zhen Wen, Fei Wang, Xiangyu Zhao, Jiliang Tang, et al. Recommender systems in the era of large language models (llms). *IEEE Transactions on Knowledge and Data Engineering*, 2024.

Technical Appendix

A FSM-SCG

Algorithm 1 FSM-SCG Algorithm Workflow

```
Input: User requirements $R$
Output: Refined smart contract $C_r$
Refer to Fig. 1 for other notations.

1: $M_{ft} \leftarrow \text{FPFT}(M_{pre}, D_{ft})$
2: $P_{R\to F} \leftarrow \text{R2FPrompt}(R)$
3: $F \leftarrow M_{ft}(P_{R\to F})$
4: while not (FormatCK($F$) and GraphCK($F$)) do
5:     $F \leftarrow M_{ft}(R)$
6: end while
7: $P_{F\to C} \leftarrow \text{F2CPrompt}(F)$
8: $C \leftarrow M_{ft}(P_{F\to C})$
9: $SuccessFlag \leftarrow$ False
10: while not $SuccessFlag$ do
11:     if not CompileCK($C$) then
12:         $P_c \leftarrow \text{CompilePrompt}(I_c)$
13:         $C \leftarrow M_{ft}(P_c)$
14:     else
15:         if not SecurityCK($C$) then
16:             $P_s \leftarrow \text{SecurityPrompt}(I_s)$
17:             $C \leftarrow M_{ft}(P_s)$
18:         else
19:             $SuccessFlag \leftarrow$ True
20:         end if
21:     end if
22: end while
23: $C_r \leftarrow C$
24: return $C_r$
```

The FSM-SCG algorithm fine-tunes a pre-trained model M_{ft} using a dataset D_{ft}, generates a Finite State Machine (FSM) F from user requirements R via a prompt $P_{R\to F}$, and validates F through format (`FormatCK`) and graph (`GraphCK`) checks in a loop until it passes. It then generates a smart contract C from F using a prompt $P_{F\to C}$, and iteratively refines C by re-generating it with compilation (`CompilePrompt`) or security prompts (`SecurityPrompt`) until it passes both compilation (`CompileCK`) and security (`SecurityCK`) checks. Finally, the refined smart contract C_r is returned after performing unit tests using a dataset D_b.

Once a valid FSM F is obtained, the algorithm generates a smart contract C by creating a prompt $P_{F\to C}$ using the `F2CPrompt` function and processing it through M_{ft}. The smart contract C undergoes further refinement through iterative validation loops. If C fails the compilation check (`CompileCK`), a compilation prompt P_c is generated using `CompilePrompt`, and M_{ft} regenerates C. Similarly, if C fails the security check (`SecurityCK`), a security prompt P_s is generated using `SecurityPrompt`, and M_{ft} regenerates C. This process continues until C passes both compilation and security checks.

Finally, the refined smart contract C_r is set to the validated C and returned as the final output, ensuring that it can pass compilation and meets security standards.

B Fine-tuning Dataset

We construct a dataset to fine-tune LLMs for smart contract generation.

Motivation and overview. Fine-tuning has been used by existing methods to enhance the performance of LLMs for smart contract generation [Liu *et al.*, 2024; Zhao *et al.*, 2024; Luo *et al.*, 2024]. However, these datasets are not public, which makes it difficult to reproduce their results. Moreover, these datasets consider a single task or scenario and thus cannot encompass the diverse requirements of practical smart contract generation. To tackle these two problems, we construct an open-source fine-tuning dataset that covers various tasks and adopts a dialogic format.

We collect smart contract source code from platforms such as Etherscan and use GPT-4o to generate the corresponding user requirements and FSM, resulting in a main dataset of approximately 30k data items. Each item consists of user requirements (R), FSM (F), and smart contract code (C). Based on this main dataset, we produce the following sub-datasets by rearranging the elements of the items. These sub-datasets can be used to train models for different tasks during smart contract generation.

- *R2F2C dataset* contains the R, F, and C. It can be used to train models that generate FSMs from user requirements and transform FSMs into smart contracts.
- *R2F dataset* contains the R and F. We train models to map user requirements to FSMs.
- *F2C dataset* contains F and C, and can be used to train models to transform FSMs into smart contracts.
- *C2F dataset* contains F and C. It allows models to derive FSMs from smart contract codes and thus understand the connection between FSM logic and contract code.
- *R2C dataset* contains R and C. We train models to directly map user requirements to smart contract codes.
- *A2C dataset* contains annotation extracted from smart contracts, along with the corresponding function code. It can be used to train models that generate function code based on these annotations.

Generation procedure. As shown in Figure 5, it takes 4 steps to generate the fine-tuning dataset.

❶ Data collecting and cleaning. We download smart contracts from sources like Etherscan and collect contract codes with varying functionalities, language versions, and complexities, which serve as raw material for constructing the dataset. For highly similar smart contracts, we retain the most complete and representative samples.

❷ Extracting FSM and requirements. We input the preprocessed smart contract code into the GPT-4o and generate the corresponding FSM and user requirements.

❸ Constructing the main dataset. We collect the code (C), FSM (F), and user requirements (R) of each smart contract

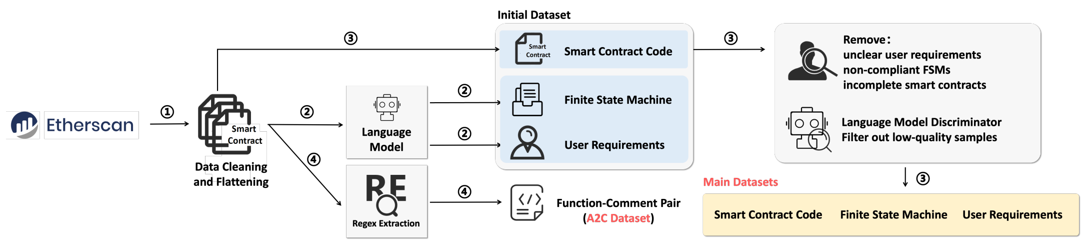
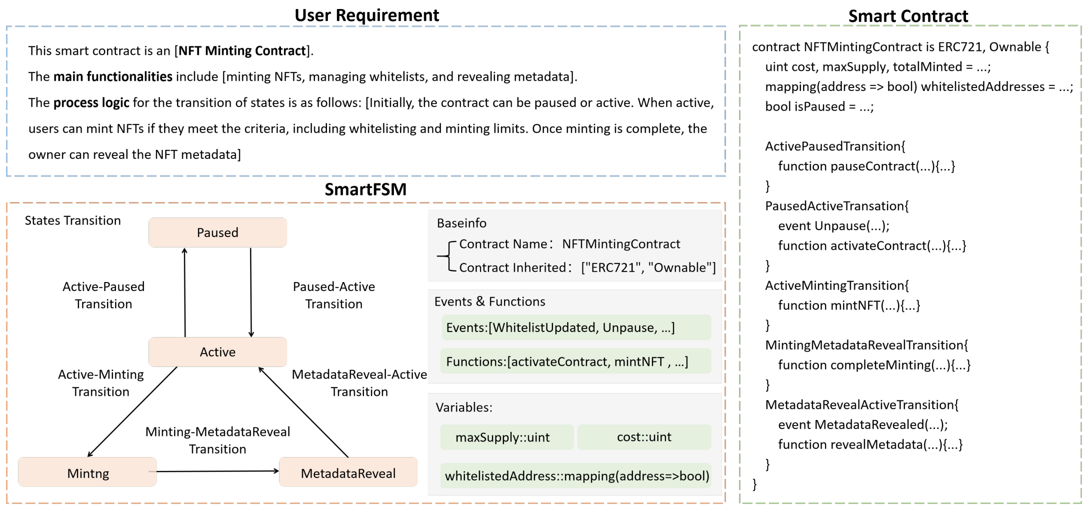


Figure 5: Procedure for generating the fine-tuning dataset.



This smart contract is an [**NFT Minting Contract**].

The **main functionalities** include [minting NFTs, managing whitelists, and revealing metadata].

The **process logic** for the transition of states is as follows: [Initially, the contract can be paused or active. When active, users can mint NFTs if they meet the criteria, including whitelisting and minting limits. Once minting is complete, the owner can reveal the NFT metadata]

Baseinfo
- Contract Name: NFTMintingContract
- Contract Inherited: ["ERC721", "Ownable"]

Events & Functions
- Events:[WhitelistUpdated, Unpause, ...]
- Functions:[activateContract, mintNFT , ...]

Variables:
- maxSupply::uint
- cost::uint
- whitelistedAddress::mapping(address=>bool)

```
contract NFTMintingContract is ERC721, Ownable {
   uint cost, maxSupply, totalMinted = ...;
   mapping(address => bool) whitelistedAddresses = ...;
   bool isPaused = ...;

   ActivePausedTransition{
      function pauseContract(...){...}
   }
   PausedActiveTransation{
      event Unpause(...);
      function activateContract(...){...}
   }
   ActiveMintingTransition{
      function mintNFT(...){...}
   }
   MintingMetadataRevealTransition{
      function completeMinting(...){...}
   }
   MetadataRevealActiveTransition{
      event MetadataRevealed(...);
      function revealMetadata(...){...}
   }
}
```

Figure 6: Three key elements involved in the smart contract generation process: Requirement, FSM, and Smart Contract.

to form the initial dataset. After manual inspection and GPT-4o scoring, the initial dataset is refined into the main dataset, with inaccurate descriptions, irregular FSMs, and incomplete smart contracts removed. Based on this main dataset, the R2F, F2C, R2F2C, C2F, and R2C subdatasets can be generated.

❹ **Annotation extraction.** We scan the preprocessed smart contract code with regular expressions for pairs of functions and annotation. This produces the A2C dataset.

C Generation Example

As shown in Figure 6, we use a specific example from the non-fungible token (NFT) minting scenario to illustrate the key elements involved in the smart contract generation process: user requirements, FSMs, and smart contract code.

D Additional Experiments

Table 5: The impact of fine-tuning methods on the effect.

Method	Model	CPR(↑)	VRS(↓)
FPFT	Qwen2.5-7B	93.6%	2.48
	LlaMa3.1-8B	95.1%	2.36
LoRA	Qwen2.5-7B	80.5%	3.37
	LlaMa3.1-8B	75.8%	3.56

Effect of fine-tuning method. As shown in Table 4, we compare FPFT and LoRA fine-tuning while fixing all other pa-

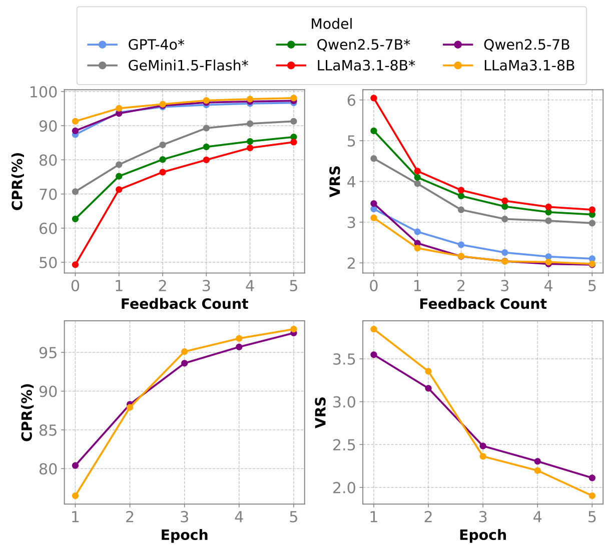


Figure 7: Comparison of model performance with respect to feedback count and training epochs. The asterisk (*) indicates that we do not fine-tune the model.

rameters. FPFT outperforms LoRA as it updates all parameters, allowing better adaptation to the target task. In contrast, LoRA freezes model weights and updates fewer parameters, which limits its adaptability to specific tasks. Moreover, since the base model has not been exposed to our target task (FSM as an intermediate representation for smart contract code generation), LoRA's more limited parameter updates result in insufficient depth in task-specific learning during training. Therefore, we choose FPFT for our experiments.

Effect of feedback count. As shown in Figure 7, we vary the feedback count of FSM-SCG from 0 to 5. When the count increases from 0 to 1, we observe the largest performance improvement. This is because most errors in the generated contracts are simple, requiring only a few iterations to correct them. In the previous experiment, we choose a feedback count of one to balance performance gain and time consumption.

Effect of fine-tuning epochs. As shown in Figure 7, we vary the fine-tuning epochs of FSM-SCG from 1 to 5. The performance improves significantly as the epoch number increases from 1 to 3, but the gain tends to diminish beyond 3. This is because, during the initial stages of fine-tuning, the model can quickly adapt to the new task and data, leading to substantial model convergence. As training progresses, the model's learning gradually saturates, leaving less room for further improvements. To balance training time and performance, we choose 3 epochs in the experiments.