ISSN: 2093-5374 / E-ISSN: 2093-5382

AI Generation of Smart Contract for Decentralized Autonomous Applications

Alfred Kuhlman[1], and Arya Wicaksana[2*]

[1]Department of Informatics, Universitas Multimedia Nusantara, Indonesia.
alfredkuhlman1@gmail.com, https://orcid.org/0009-0007-5150-3838

[2*]Department of Informatics, Universitas Multimedia Nusantara, Indonesia.
arya.wicaksana@umn.ac.id, https://orcid.org/0000-0002-0888-036X

Received: November 11, 2024; Revised: December 20, 2024; Accepted: February 11, 2025; Published: March 31, 2025

Abstract

Blockchain and smart contracts enable the development of Decentralized Autonomous Systems (DASs), such as Decentralized Autonomous Applications (DAAs) and Decentralized Finance (DeFi). This paper addresses the challenge of automating smart contract generation for DASs by leveraging the Code LLaMA – Instruct model. A dataset of 6,003 human instruction and source code pairs is used for fine-tuning, employing Quantized Low-Rank Adaptation (QLORA) to optimize the model's seven billion parameters. The study focuses on generating Solidity-based smart contracts for Ethereum, evaluating the model across four scenarios: cryptocurrency token creation, ownership management, DAO wallet whitelisting, and company information contracts. Results indicate that the fine-tuned model successfully generates functional and efficient smart contracts, demonstrating correctness and optimized gas usage.

Keywords: Autonomous, Blockchain, Code Generation, Decentralized System, Smart Contract.

1 Introduction

The advancement of blockchain and smart contract technologies provides a foundation for the creation of Decentralized Autonomous Applications (DAAs), e.g., Decentralized Autonomous Applications (DAAs) (Augustin et al., 2023; Santana & Albareda, 2022; Zhao et al., 2022) and Decentralized Finance (DeFi) (Chen & Bellavitis, 2020; Zetzsche et al., 2020). Users in DAAs participate and contribute actively in managing and running the system together without relying on a single trusted entity. The entire operation of a DAA can be carried out using smart contracts on top of a blockchain platform. However, creating and deploying a smart contract still relies on developers, which introduce design and implementation flaws (Chen et al., 2020), inefficiency in gas usage (Nelaturu et al., 2023), and security vulnerabilities (Chu et al., 2023; Hassooni, 2024) in addition to limiting users without coding and technical background to participate and contribute to the system directly without intermediaries of smart contract developers.

Despite their potential, smart contract development presents significant challenges. Writing and deploying smart contracts requires technical expertise, creating a barrier for non-programmers who wish to participate in decentralized systems. Human errors in contract design can lead to inefficiencies in gas usage, security vulnerabilities, and unintended logic flaws. In the context of DAAs, smart contract gen-

Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA), volume: 16, number: 1 (March), pp. 456-477. DOI: *10.58346/JOWUA.2025.I1.027*

*Corresponding author: Department of Informatics, Universitas Multimedia Nusantara, Indonesia.

eration involves multiple steps, including proposal submission, voting, and contract deployment, often requiring extensive off-chain and on-chain coordination. These complexities highlight the need for automated smart contract generation to streamline development, enhance accessibility, and minimize errors (Santana & Albareda, 2022; Zhao et al., 2022).

The breakthrough of Large Language Models (LLMs) enables the generation of text and code based on human language within a conversation, e.g., GPT-4 (Achiam et al., 2023), LLaMA 2 - Chat (Touvron et al., 2023), PaLM 2 (Anil et al., 2023), FLAN-T5 (Chung et al., 2024), WizardLM (Xu et al., 2023) to code generation models which are WizardCoder (Luo et al., 2023), CodeT5 (Wang et al., 2021), CodeBERT (Feng et al., 2020), and the state-of-the-art model for code generation, the Code Llama (Roziere et al., 2023). These models have shown their capabilities to answer human prompts and be usable for various tasks. AI-generated smart contracts in a DAA maintain decentralization and autonomous characteristics of the system and reduce human errors and limitations in developing smart contracts (Meinhardt & Krein, 2024; Hassooni, 2024). Furthermore, users with no coding background can generate smart contracts directly using the technology.

This paper proposes the use of the Code LLaMA – Instruct model for automated smart contract generation in decentralized systems. The focus is on Solidity, the dominant programming language for Ethereum-based smart contracts. The key contributions of this study include:

1. A human instruction dataset tailored for Solidity smart contract generation.
2. A fine-tuned Code LLaMA – Instruct model for automatic smart contract creation.
3. A demonstration of AI-generated smart contracts within Decentralized Autonomous Applications (DAAs).
4. An evaluation of the generated contracts in terms of functionality, gas efficiency, and security.

The rest of this paper is structured as follows: Section 2 provides background information and related work on smart contracts, AI code generation, and evaluation methodologies. Section 3 details the methodology, including data collection, system design, and testing. Section 4 presents experimental results and analysis across four smart contract scenarios. Finally, Section 5 concludes with key findings and future research directions.

2 Preliminaries

Related Works

Allouche et al. proposed a method to generate smart contracts for the Internet of Media Things (IoMT) by utilizing IoMT Application Programming Interfaces (APIs) and smart contract prototypes (Allouche et al., 2021). The architecture involves two main actors: an IoMT expert who makes IoMT applications and a blockchain expert who supplies smart contract prototypes for use (Sadulla, 2024). The system has three main components: IoMT parser to extract relevant information for blockchain use, smart contract developer to combine the smart contract prototype and IoMT specifications received, and blockchain manager to deploy the finished smart contract.

The following year, Marchesi et al. published a paper to automate Ethereum-based smart contract generation in the agri-food domain (Marchesi et al., 2022). The approach that was used was modular and configurable smart contract templates. The process starts with creating templates from blockchain experts for each possible use case. These templates can then be combined into one functional smart contract. Users must give input in JavaScript Object Notation (JSON) or Comma-Separated Values

(CSV) files consisting of actors, events, producers, products, and resource data to generate the smart contract as accurately as the user wants.

Liu et al. propose to design smart trade application contracts by transforming the finite state machine (FSM) model (Zain, 2025; Liu et al., 2022). The FSM model has to capture the flow of data, actors, and other parameters needed to generate an accurate smart contract. One main problem with FSM is its possibility of a state explosion. State explosion is mitigated using Discrete Events and Hierarchical State Machine (DE-HSM) models as a safe alternative.

On LLM, (Karanjai et al., 2023) conducted an empirical study on generated smart contracts from ChatGPT and PaLM 2. The study shows that these LLMs can generate smart contracts with varying success (Kamaraj & Amudha, 2017). These LLMs need to be trained specifically to generate smart contracts, and as a result, over 70% of the LLMs' responses need to be compilable using their prompt template. Security vulnerabilities with high severity rating categories are found in these smart contracts using Slither.

The works in (Allouche et al., 2021) and (Marchesi et al., 2022) use premade templates that require experts to create smart contract templates. The work in (Liu et al., 2022) with the FSM model requires some knowledge of FSM to generate the desired smart contracts. LLMs in (Karanjai et al., 2023) can generate smart contracts without needing a smart contract template and expert. Thus, this paper further extends the approach of using LLM to auto-generate smart contracts that are correct, efficient, and secure for DAAs.

Smart Contract

A smart contract is a computer program that can be executed when the program requirements are met. These smart contracts are immutable and tamper-proof, distributed through the blockchain network. Its decentralized nature allows these smart contracts to be used without requiring trusted intermediaries, making them transparent, secure, and trustless. Smart contract development challenges are readability, functionality, execution efficiency, contract correctness, privacy, and security (Zheng et al., 2020).

Smart contracts executed on blockchain platforms such as Ethereum consume gas as the fee for the computation it runs. These gas fees are equal to real-world currency which values can varies depending on platform and time. Gas is usually charged on every transaction or execution of the smart contract. The Ethereum Yellow Paper described the breakdown cost of gas for smart contract transactions and execution (Wood, 2014). Gas costs are based on the computational operation in the smart contract function; meaning two smart contracts that behave the same can have different gas cost because of how they were written. For example, incrementing a number using double plus sign (++) costs more gas if it was written in postfix instead of prefix. Thus, writing smart contract codes has to take gas fees into account.

Smart contracts developed for Ethereum in Solidity codes can be vulnerable to exploitation by malicious actors. Well-known attacks are reentrancy, which allows an attacker to initiate unintended loops generally due to improper check for malicious fallback function call, overflow and underflow of variable values done by an attacker with inputs exceeding the minimum or maximum value it can take, and default visibility attack that happens when the developer does not set private visibility on a function that should have (Sayeed et al., 2020). One example is a reentrancy attack that happens in 2016 on The DAO which made the smart contract to withdraw its funds to the hacker. Because of danger of malicious actors, the immutability nature of smart contracts, and the complication of fixing deployed faulty smart contracts, it is crucial to test and verify them before deployment (Jiao et al., 2020).

The static analysis tool is a way to detect vulnerability in smart contracts by reading the smart contract and matching any part of the code with patterns attributed to vulnerability. The code is not executed, unlike symbolic execution analysis. Therefore, the automation is straightforward and does not require test cases to be written beforehand. One major issue with the current analysis tools is that they can only detect previously known vulnerabilities (Ghaleb & Pattabiraman, 2020).

Code Llama - Instruct

The LLaMA 2 paper presents two model variants: the pre-trained (LLaMA 2) and the fine-tuned for conversation (LLaMA 2 - Chat). Using new and increased corpus data for training, doubled context length from LLaMA, and grouped-query attention architecture in (Ainslie et al., 2023) for training, LLaMA 2 outperformed most open-source models. The fine-tuned model, LLaMA 2 - Chat, is created using Reinforcement Learning with Human Feedback (RLHF), where the model is directed to respond according to the human annotator's preference through a reward model (Touvron et al., 2023).

A new open-source LLM for coding purposes: Code Llama. Code Llama is derived from LLaMA 2 with a 7 billion, 13 billion, and 34 billion parameters model and trained on 500 billion token data for infilling code tasks, which means filling in the blanks of source codes. The model went through a specialization phase, as seen in Table 1. In the specialization phase, these three variants were fine-tuned for long-context on 20 billion token data to take in longer human inputs (Feist et al., 2023).

Table 1: Comparison of Code Llama Models

Model	Purpose	Specialization
Code Llama	Code completion	Long context fine-tuning with 20 billion tokens
Code Llama - Instruct	Question-answer	Long context fine-tuning with 20 billion tokens and instruction fine-tuning with 5 billion tokens
Code Llama - Python	Python programming	Python code training with 100 billion tokens and long context fine-tuning with 20 billion tokens

Code Llama - Instruct usage is similar to the other Code Llama variants. The model takes in text inputs and later encodes them into tokens to predict the following output tokens. These output tokens are decoded back to readable text as the final output. Code Llama and its variants can generate output with up to 100,000 tokens of context and take in input up to 100,000 tokens in length. By having a context of up to 100,000 tokens, Code Llama can generate output from long inputs and past outputs more accurately (Feist et al., 2023).

The fine-tuning process requires a dataset for the model's existing parameters to be retrained. Originally, a complete fine-tuning process for an LLM with billions of parameters requires a hundred gigabytes of GPU memory. A better technique to fully fine-tune LLMs was proposed using Quantized Low-Rank Adaptation (QLoRA), which improved the Low-Rank Adaptation (LoRA) paper. LoRA enables full fine-tuning of an LLM with a smaller memory footprint by training on smaller model parameters without hindering the model's performance (Hu et al., 2022).

QLoRA improved it further by quantizing the base model to reduce memory footprint when loading the model, using a paged optimizer to prevent memory spikes when training by CPU offloading, double quantization, and 4-bit Normal Float (NF4) data type to lessen the memory footprint further. With these innovations, QLoRA can fully fine-tune LLMs with around 17 times smaller memory footprint than the original method without losing the model's overall performance (Dettmers et al., 2023).

Evaluation Metrics

The evaluation metrics for the AI-generated smart contracts are code correctness, gas consumption (efficiency), and number of vulnerabilities (security). User testing and comparison with the reference smart contract can evaluate the code's correctness. The efficiency of the generated smart contract can be evaluated by the gas consumption through deployment on the Ethereum blockchain network or using the Remix IDE (Jain, 2022) or Truffle framework (Jain, 2022). The security of the generated smart contracts can be evaluated by the number of vulnerabilities found by security analysis tools, e.g., Slither (Feist et al., 2019) and Manticore (Mossberg et al., 2019).

There has yet to be a consensus on the best way to evaluate code correctness, as smart contract structure can vary for different developers. One method used in LLM evaluations is a benchmarking dataset of tasks and corresponding answers. The score is measured on how well the model generates the best answer (Peng et al., 2023; Yuan et al., 2023). For AI-generated smart contracts, there has not been an open benchmarking dataset for evaluating an instruct model.

Smart contract gas consumption depends on the code implementation and execution of every transaction. These costs can be checked with Remix IDE or Truffle framework. Evaluation is done by measuring the gas transaction cost for each function in the smart contract (Zhang & Lee, 2019).

Security evaluation of a smart contract is possible using Slither and Manticore by summing all the detected vulnerabilities according to vulnerability types (Dingman et al., 2019; Tikhomirov et al., 2018; Ye et al., 2019). These vulnerability types are re-entrance, overflow, underflow, transaction origin usage, and self-destruct smart contracts (Almakhour et al., 2020).

Slither defines the vulnerability levels by three severity levels, which are low, medium, and high. Aside from vulnerability, Slither detects non-security-related code patterns if optimizations or informational recommendations are needed. Each vulnerability and non-security type requires a detector, and Slither has 94 documented detectors with varying detection confidence levels (Slither, the smart contract static analyzer, 2023).

3 Methods

Dataset Collection and Preprocessing

Fine-tuning Code Llama - Instruct model for the smart contract auto-generation requires a dataset of human prompt text and its corresponding answer. For this purpose, the required dataset is obtained from the GPT-3.5 turbo and GPT-4, and the corresponding smart contract source code from the Solidity source code dataset. We use GPT to generate model prompts is because the data obtained from this method can improve the overall performance while keeping the data privacy-compliant (Dabre et al., 2024; Gholami & Omar, 2023; Liu et al., 2024). The Solidity source code dataset used is available in the HuggingFace platform with the dataset title "Slither Audited Smart Contracts" by Martina Rossini (Slither Audited Smart Contracts Dataset, 2022). The general steps to create the dataset can be seen in Figure 1 which steps are as follows.

1. Gather Solidity source code from the "Slither Audited Smart Contracts" dataset.
2. Preprocess the source code to replace three or more new line characters with two new line characters.
3. Put the pre-processed source code into an instruction template.
4. Call the GPT API with the string instruction as its prompt.

5. Collect the GPT model responses.
6. Clean up the Solidity source code further by deleting irrelevant comments.

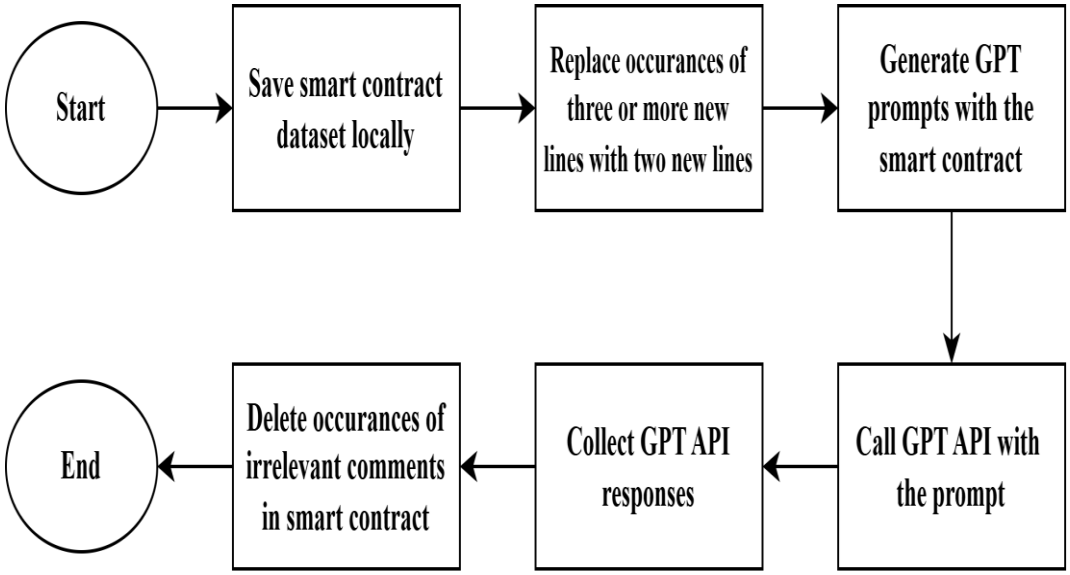

Figure 1: Data Collection and Preprocessing Flow

The source code gathering steps started with saving the “Slither Audited Smart Contracts” dataset in a local storage for it to be processed easier. Before calling GPT API to generate a human prompt, the saved source codes are preprocessed by replacing three or more new line characters with two new line characters using regex. This preprocess step is done to ensure the model's output is more consistent by not outputting three or more new line characters on its output. Once the source codes have been preprocessed, these source codes are put into an instruction template as prompts for the GPT models. This template is used every time the GPT API is called.

After the generated human prompts have been gathered, the source code is processed by deleting irrelevant comments, usually positioned at the start of the smart contract, and having the phrase "Submitted for verification." These smart contracts are processed to remove the comment using regex to delete the irrelevant comments that the regex cannot delete.

The final dataset consists of 6,003 data with three columns: human instruction as the smart contract requirement, subsequent Solidity source code, and the GPT model used to generate the human instruction. From the gathered data, distributions of the GPT model used can be seen in Table 2.

Table 2: Collected Dataset from the GPT-3.5 and GPT-4

LLM	**Amount**
GPT-3.5 Turbo	5,276
GPT-3.5 Turbo 16k context	678
GPT-4	49

Design and Implementation

The requirements and design specification for the input, process, and output of the finetuned Code Llama - Instruct to work as intended can be seen in Table 3.

Table 3: Comparison of Code Llama Models

Feature	Requirement
Input prompt should be a sentence or a paragraph of smart contract requirements at the start of the text "Make a smart contract to [input smart contract purpose here]".	In DAA, the process begins with proposal submission. This proposal contains new features or rules to be added to the DAA. Starting the text prompt as mentioned ensures the smart contract purpose is clearly stated first.
The model's context length follows Code Llama - Instruct, up to 100,000 tokens.	It allows users to obtain longer and more complex smart contract codes
The model output is Solidity source code in text form streamed per token or all at once.	The output is a smart contract in the Solidity programming language.

The application's design and fine-tuning of the Code Llama – Instruct model follows the specification requirements. The model should be able to be loaded locally or through the cloud. Once the model is loaded, the user can input their prompt to generate a smart contract using the instruction template in Table 4. The user prompt is then encoded into tokens for the LLM to generate a response. The LLM generates a tokenized response, which is then decoded back for the user to see. The smart contract generation process can be seen in Figure 2.

Table 4: User Text Prompt for Smart Contract Auto-Generation

Text Prompt
Instruction: Use the Task below and the Input given to write the Response, which is a programming code that can solve the following Task: ### Task: [Human Instruction] ### Solution: [Source Code]

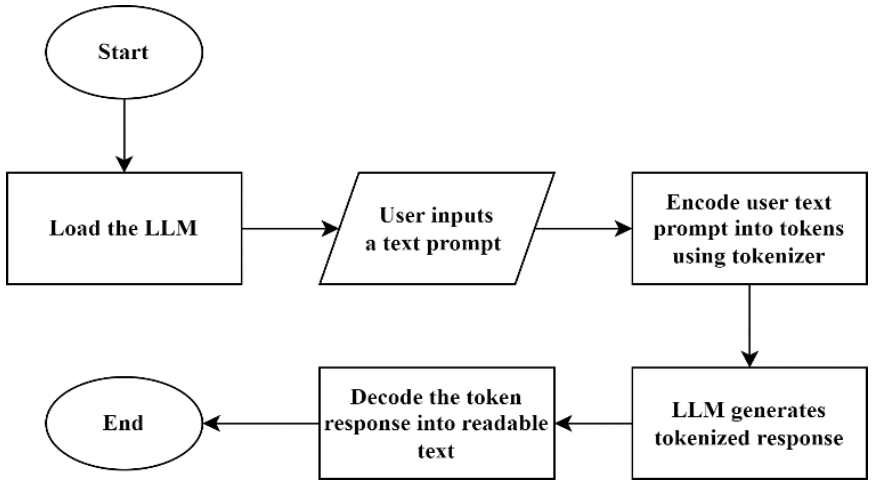

Figure 2: Data Collection and Preprocessing Flow

After collecting the smart contract instruction dataset, the Code Llama - Instruct model can be fine-tuned to generate smart contracts based on user prompts. The fine-tuning process was conducted using one NVIDIA GeForce GTX 1080Ti GPU. Due to processing limitations, the 7 billion parameters model is used as the base model. The model is fine-tuned using the QLoRA method provided by Parameter-Efficient Fine-Tuning (PEFT) library (Mangrulkar et al., 2022). The procedure to fine-tune the Code Llama - Instruct model follows.

1. Load the smart contract instruction dataset.
2. Initialize instruction template for training.
3. Load the Code Llama - Instruct model with bits and bytes quantization for memory efficiency.
4. Set up a supervised fine-tuning trainer to train the Code Llama - Instruct model.
5. Begin training the model parameters with the instruction template.
6. Merge the trained model parameters with Code Llama - Instruct model parameters.

The parameters used to train the model are based on a fine-tuned LLaMA 2 for Python instructions by Eduardo Muñoz (Llama-2-7b-int4-python-coder, 2023) with some tweaking to ensure the training works on our machine. These parameters are used for bitsandbytes and supervised fine-tuning trainer library. Bitsandbytes library loads the Code Llama - Instruct model memory efficiently. The bitsandbytes quantization configuration can be seen in Table 5. The supervised fine-tuning trainer library trains the loaded Code Llama - Instruct model. PEFT has LoRA configurations based on the QLoRA paper. The training parameters can be seen in Table 6. The generated dataset combines the human prompt and its source code pair into a single string to train the model.

Table 5: Bitsandbytes Quantization Configuration

Name	Value
Load in 4-bit	True
4-bit quantization type	Code completion
4-bit compute type	Float16
4-bit use double quantization	True

Table 6: Training Parameters

Name	Value
Number of train epoch	1
FP16	True
FP16 option level	O1
BF16	False
Per device train batch size	1
Gradient accumulation steps	1
Gradient checkpointing	1
Max gradient normal	0.3
Learning rate	2e-4
Weight decay	0.001
Optimizer	Paged AdamW 32-bit
Learning rate scheduler type	Cosine
Warmup ratio	0.03

The training is set with one epoch and one batch size using all 6,003 data from the dataset once. On every 100 steps, the training loss is observed to find out the model's learning capability. The training took around 21 hours, 4 minutes, and 57 seconds to finish. The graph of the training loss can be seen in Figure 3. The trend line in the training loss graph goes down, indicating that the model has learned from the dataset to generate a smart contract. The resulting finetuned model is saved and published to HuggingFace for easier access for testing and evaluation.

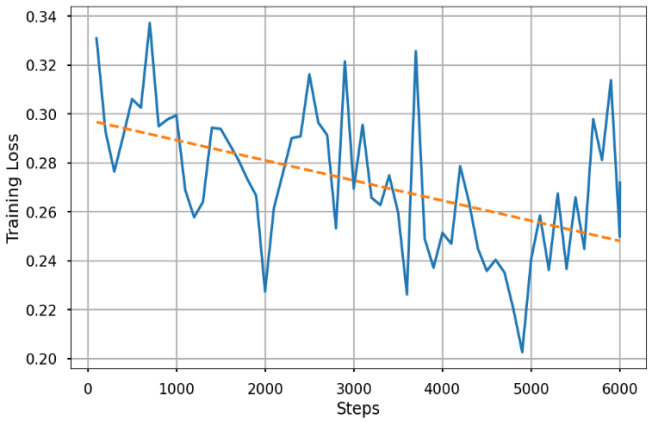

Figure 3: Training Loss in the Fine-Tuning Process

Testing and Evaluation

The model is put into a task to solve four different scenarios. Each scenario is based on an existing smart contract and will be compared with the generated smart contract to see how well it performs. The generated smart contract will be evaluated through unit testing, gas cost comparison, and slither vulnerability check comparison to know how well it performs for code correctness, gas efficiency, and vulnerability.

The evaluation of the produced fine-tuned model uses four code samples from the Slither Audited Smart Contracts dataset that are not used in the instruction dataset. These code samples are picked based on their use case to ensure they are not similar. Then, a human prompt to generate the smart contract is created using GPT3.5-turbo with the same prompt when making the instruction dataset (prompt can be seen in Table 4). The fine-tuned model then uses the created prompt to generate the smart contract. If the generated smart contract is irrelevant to the original smart contract, manually tweak the prompt until it is relevant. The process for smart contract generation for test scenarios can be seen in Figure 4.

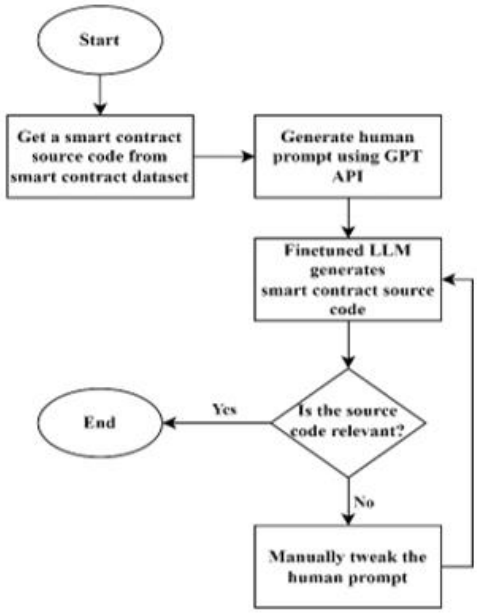

Figure 4: Smart Contract Auto-Generation Test Flow

The unit testing criteria are determined based on how the referenced smart contract works. Remix IDE is used to conduct unit testing and gas cost comparison. The security of the smart contracts is evaluated using Slither. In Remix IDE, the AI-generated and the reference smart contracts are compiled and deployed on Remix Virtual Machine (VM). These deployed smart contracts are tested and monitored for gas cost on every transaction. The testing and evaluation process can be seen in Figure 5.

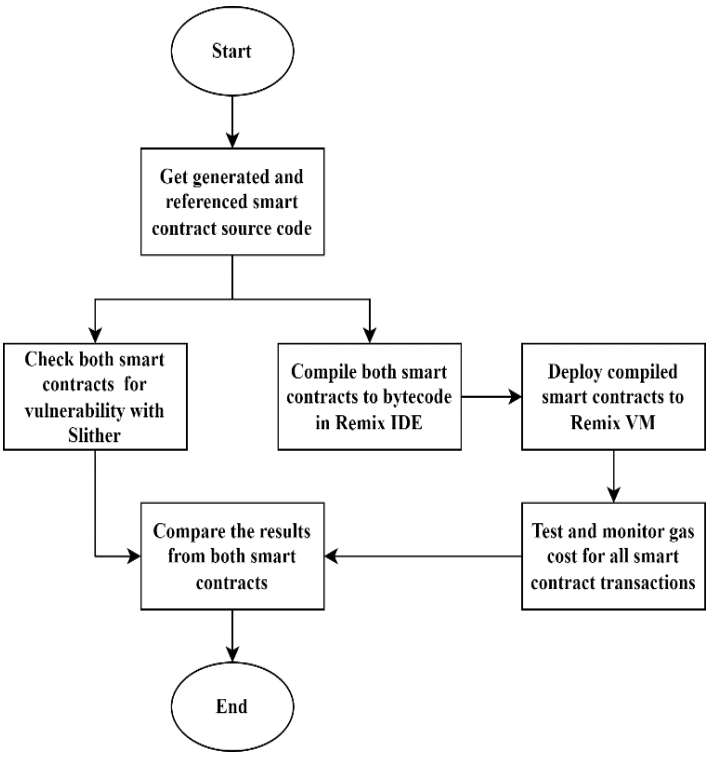

Figure 5: Testing and Evaluation Flow

Four scenarios are developed for evaluating the fine-tuned model: cryptocurrency token generation, ownership smart contract, DAO wallet whitelisting, and company information smart contracts. Each scenario uses one T4 GPU to generate the smart contract. The fine-tuned model has the same quantization configuration as the fine-tuning process. Parameters used to generate smart contracts are based on the fine-tuned LLaMA 2 for Python instructions model with tweaks in temperature for repeatability. The model code generation parameters can be seen in Table 7.

Table 7: Parameters for Smart Contract Auto-Generation

Name	Value
Use sampling	True
Temperature	0.01
Top P	0.9

4 Results and Analysis

Cryptocurrency Token (PhoenixCrypto)

In this scenario, a user wants a smart contract to distribute their cryptocurrency token named "PhoenixCrypto" and "PXT" as its symbol. The smart contract criteria are as follows.

1. Allows the current user to transfer their token to another user.
2. Any user can set a token allowance amount for another user to receive the token from them.
3. Any user that got a token allowance from another user can use the allowed tokens themselves or transfer them to another user.
4. Any user can check their current token amount.
5. Any user can check how many token allowances two users have.
6. The transaction is rejected if any user sends a token more than they have or the allowed token amount.

The text prompt is created for the model from the smart contract reference: [Make a smart contract to create a cryptocurrency called "PhoenixCrypto" with the symbol "PXT". The contract should allow users to transfer tokens, check their balance, and approve other addresses to spend tokens on their behalf. The total supply of tokens is set to 100000000.]. The model took 47 seconds to generate the smart contract.

The generated smart contract and its referenced smart contract were compiled into bytecode. The bytecode size of the generated smart contract is smaller than the referenced, which is 6,286 bytes for the generated smart contract and 8,556 bytes for the referenced smart contract. The compiled smart contracts are deployed in the Remix VM environment, and all transactions are tested to check if the smart contract criteria are met. The unit testing result can be seen in Table 8. The generated smart contract can satisfy all the criteria. Gas consumption for transaction and execution costs on all the smart contract transactions are monitored and can be seen in Table 9. The table does not include any call functions and functions that cannot be executed. From the test, the referenced smart contract costs more to deploy than the generated smart contract.

Table 8: Unit Testing of PhoenixCrypto

Number	Criteria	Pass?
1	Allows the user to transfer their token to another user.	Yes
2	Any user can set a token allowance amount so another can receive tokens from them.	Yes
3	Any user that got a token allowance from another user can use the allowed tokens to themselves or to transfer to another user.	Yes
4	Any user can check their current token amount.	Yes
5	Any user can check the amount of token allowances between any two users.	Yes
6	The transaction is rejected if any user sends a token that is more than they have or the allowed token amount.	Yes

Table 9: Gas Cost Comparison of PhoenixCrypto

Function	Transaction Cost		Execution Cost	
	Reference	Generated	Reference	Generated
Contract deployment	974,267	748,533	854,891	647,147
Approve	46,051	45,990	24,479	24,418
Transfer	51,592	51,395	30,020	29,823
Transfer from	52,524	57,233	35,384	35,293

The generated smart contract and smart contract reference are checked on Slither for vulnerability. The generated smart contract analysis result can be seen in Table 10. Six unique results were found from the generated smart contract, which are as follows.

1. Incorrect versions of Solidity (informational).
2. Conformance to Solidity naming conventions (informational).

3. Conformance to numeric notation best practices (informational).
4. State variables that could be declared constant (optimization).

The security analysis from Slither found three unique results from the smart contract reference, which are as follows.

1. Incorrect versions of Solidity (informational).
2. Conformance to Solidity naming conventions (informational).
3. Conformance to numeric notation best practices (informational).

Table 10: Security Analysis on PhoenixCrypto

Smart Contract	Other	Low	Medium	High
Reference	3	0	0	0
Generated	4	0	0	0

Ownership Smart Contract

In this scenario, a user needs an ownership system that can be implemented as a library for other existing smart contracts. The ownership system will help implement user permission. The smart contract criteria are as follows.

1. Allow the current owner to transfer their ownership to another user address.
2. Any user can check the current owner.
3. Allow the current owner to renounce the smart contract.
4. Non-owners should not be able to transfer ownership or renounce the smart contract.

The smart contract reference creates a text prompt for the model: [Make a smart contract to implement basic authorization control functions, allowing the contract owner to transfer ownership and relinquish control. This smart contract's purpose is to simplify user permissions implementation by ensuring that only the owner can call certain functions.]. The model took 38 seconds to generate the smart contract. The generated smart contract and its reference are almost identical, with the difference in the modifier and smart contract version.

The generated smart contract and its referenced smart contract were compiled into bytecode. The bytecode size of the generated smart contract is less than the referenced, which is 2,114 bytes for the generated smart contract and 2,686 bytes for the referenced smart contract. The compiled smart contracts are deployed in the Remix VM environment, and all transactions are tested to check if the smart contract criteria are met. The unit testing result can be seen in Table 11. The generated smart contract has been shown to meet the smart contract criteria. Gas consumption for transaction and execution costs on all the smart contract transactions are monitored and can be seen in Table 12. The generated smart contract has a noticeably smaller contract deployment cost.

Table 11: Unit Testing of Ownership Smart Contract

Number	Criteria	Pass?
1	Allows the current owner to transfer their ownership to another user address.	Yes
2	Any user can check the smart contract owner.	Yes
3	Allow the current owner to renounce the smart contract.	Yes
4	Non-owners should not be able to transfer ownership or renounce the smart contract.	Yes

Table 12: Gas Cost Comparison of Ownership Smart Contract

Function	Transaction Cost		Execution Cost	
	Reference	Generated	Reference	Generated
Contract deployment	344,325	283,708	271,853	214,604
Renounce ownership	22,937	22,937	6,673	6,673
Transfer ownership	28,670	28,670	7,238	7,238

The generated smart contract and its reference are checked on Slither for vulnerability. The generated smart contract analysis result can be seen in Table 13. Two unique results were found for the generated and the referenced smart contract.

1. Incorrect versions of Solidity (informational).
2. Conformance to Solidity naming conventions (informational).

Table 13: Security Analysis on Ownership Smart Contract

Smart Contract	Other	Low	Medium	High
Reference	2	0	0	0
Generated	2	0	0	0

DAO Wallet Whitelister

In this scenario, a DAO owner wants a smart contract to document user wallets that are allowed for their organization. The owner allows an address to be used as a wallet checker. As the DAO holder can change occasionally, control over the owner and its users is required. The smart contract criteria are as follows.

1. A smart contract can be deployed with its owner address attached other than the deployer.
2. The current owner can approve or revoke a specific wallet address.
3. The current owner can switch its owner role to another wallet address.
4. The current owner can set an address as a wallet checker.
5. Users can check if they are approved, the current owner's address, and the current wallet checker.
6. Non-owners should not be able to approve, revoke, switch owners, or set check wallet addresses.

The smart contract reference creates a text prompt for the model: [Make a smart contract to create a whitelist of approved wallets. The purpose of this contract is to allow the DAO (Decentralized Autonomous Organization) to approve or revoke certain wallets and set a checker address for additional validation if needed. The current owner can change the current owner's address.]. The model took 22 seconds to generate the smart contract.

The generated smart contract and its referenced smart contract were compiled into bytecode. The bytecode size of the generated smart contract is less than the referenced, which is 3,570 bytes for the generated smart contract and 6,068 bytes for the referenced smart contract. The compiled smart contracts are deployed in the Remix VM environment, and all transactions are tested to check if the smart contract criteria are met. The unit testing result can be seen in Table 14. The generated smart contract does not allow the owner input address to be set when deploying the smart contract. However, its owner can be changed afterward using the function to change owner. Gas consumption for every transaction and execution cost are monitored. The gas consumption for transaction and execution can be seen in Table 15. The resulting gas consumption for the generated smart contract is less than the referenced smart contract.

Table 14: Unit Testing of DAO Wallet Whitelister

Number	Criteria	Pass?
1	Smart contracts cannot be deployed using wallet addresses other than the owner.	Yes
2	The current owner can approve or revoke certain wallet addresses.	Yes
3	The current owner can switch its owner role to another wallet address.	Yes
4	The current owner can set an address as an address checker.	Yes
5	Users can check if they have been approved, including the current owner's address and the current address checker.	Yes
6	Non-owners should not be able to approve, revoke, switch owners, or set a check wallet address.	Yes

Table 15: Gas Cost Comparison of DAO Wallet Whitelister

Function	Transaction Cost		Execution Cost	
	Reference	Generated	Reference	Generated
Contract deployment	556,799	440,860	467,841	360,340
Approve	47,173	46,216	25,741	24,784
Revoke	25,317	24,184	8,685	7,552
Set checker	47,262	46,104	25,830	24,672
Change owner	28,184	26,960	6,752	5,528

The generated smart contract and smart contract reference are checked on Slither for vulnerability. The generated smart contract analysis result can be seen in Table 16. Four unique results were found for the generated smart contract as follows.

1. Missing events access control (low severity).
2. Missing zero address validation (low severity).
3. Incorrect versions of Solidity (informational).
4. Conformance to Solidity naming conventions (informational).

Security analysis using Slither found four unique results from the smart contract reference, which are as follows.

1. Missing zero address validation (low severity).
2. Incorrect versions of Solidity (informational).
3. Missing inheritance (informational).
4. Conformance to Solidity naming conventions (informational).

Table 16: Security analysis on DAO Wallet Whitelister

Smart Contract	Other	Low	Medium	High
Reference	3	1	0	0
Generated	2	2	0	0

Company Information Smart Contract

This scenario requires a smart contract to store the company's information, i.e., the company's name, site, email, phone number, and other information. The smart contract owner can add and change additional information and self-destruct the smart contract. The smart contract criteria for this scenario are as follows.

1. Only the smart contract owner can set company information.

2. The smart contract owner can destroy the smart contract.
3. Users can retrieve the company's basic information: name, site, email, and phone number.
4. Any user can retrieve the company's additional information.

The smart contract reference creates a text prompt for the model: [Make a smart contract to store and retrieve company info. The purpose of this contract is to allow the contract owner to set and get additional information related to the company, such as bytes32 to string key-value pairs. The smart contract can self-destruct. The smart contract is named DAppsDevs. Company information is retrievable, which includes the company name (DApps Devs LLC), company site (dappsdevs.io, dappsdevs.com), phone number (+1-302-481-9195), and company email (info@dappsdevs.com).]. The model took 21 seconds to generate the smart contract. The generated smart contract has a different implementation for storing the company's information by storing all its basic information in a map, unlike the reference, which individually stores them.

The generated smart contract and its referenced smart contract were compiled into bytecode. The bytecode size of the generated smart contract is less than the referenced, which is 4,736 bytes for the generated smart contract and 5,236 bytes for the referenced smart contract. The compiled smart contracts are deployed in the Remix VM environment, and all transactions are tested to check if the smart contract criteria are met. The unit testing result can be seen in Table 17. The generated smart contract did achieve all the criteria. Gas consumption for transaction and execution costs on all the smart contract transactions are monitored. The gas consumption for transaction and execution can be seen in Table 18. The generated smart contract deployment cost is smaller than the smart contract reference.

Table 17: Unit Testing of Company Information Smart Contract

Number	Criteria	Pass?
1	Only a smart contract owner can set company information.	Yes
2	The smart contract can destroy the smart contract.	Yes
3	Any user can retrieve the company information: name, website, email, and phone number.	Yes
4	Any user can retrieve the company additional information.	Yes

Table 18: Any User Can Retrieve the Company Additional Information

Function	Transaction Cost		Execution Cost	
	Reference	Generated	Reference	Generated
Contract deployment	620,211	517,781	527,103	430,069
Self-destruct	28,553	28,575	7,489	7,511
Set company information	46,991	46,969	25,139	25,117

The generated smart contract and smart contract reference are checked on Slither for vulnerability. The generated smart contract analysis result can be seen in Table 19—the generated smart contract and the smart contract reference results in the same amount, as follows.

1. Incorrect versions of Solidity (informational).
2. Public function that could be declared external (optimization).

Table 19: Security Analysis on Company Information Smart Contract

Smart Contract	Other	Low	Medium	High
Reference	2	0	0	0
Generated	2	0	0	0

After four scenarios were tested, the performance of the fine-tuned model on each of the scenarios can be compared. The graph on total gas cost and security issues found can be seen in Figure 6 and Figure 7. With how the fine-tuned model performs on unit testing, gas cost analysis, and security analysis, key points can be derived in Table 20.

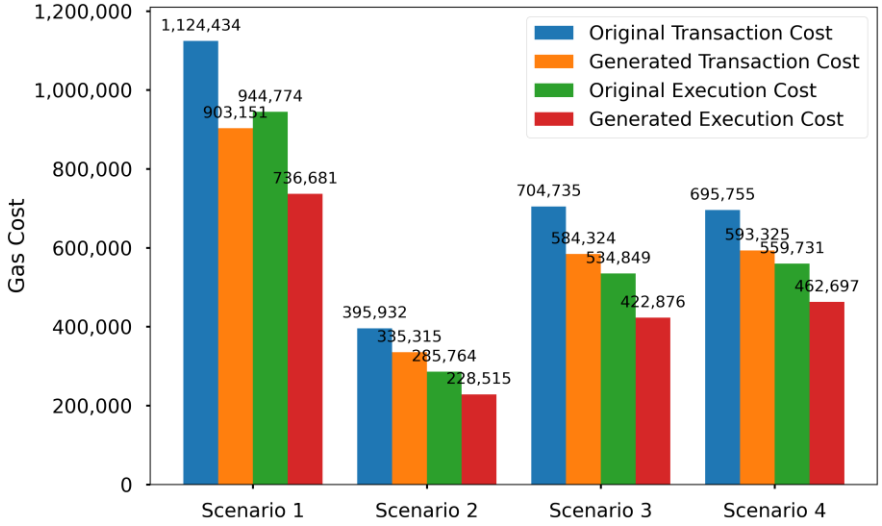

Figure 6: Total Cost Smart Contract

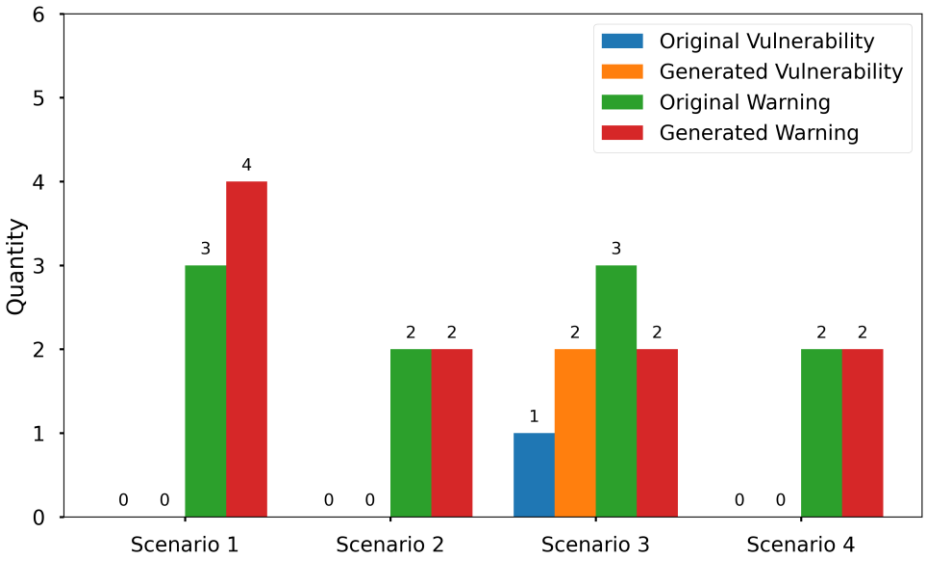

Figure 7: Total Security Issues Found by Slither

Table 20: Result Evaluation Summary

Evaluation Metric	Summary
Accuracy	The fine-tuned model can generate smart contract according to user requirements in the given four scenarios.
Gas cost	In all of the cases, the generated smart contracts costs less gas in transaction and execution.
Security	The generated smart contracts have the same amount or more vulnerability to attacks than the original smart contract.

5 Discussion

Smart contract auto-generation eliminates the need for a middleman in smart contract development, which is crucial in decentralized autonomous applications (DAAs), where decentralization and automation are the two most important characteristics. By leveraging large language models (LLMs) such as Code Llama – Instruct, this study demonstrates the potential of AI-driven smart contract development in minimizing human involvement while maintaining functional integrity. The use of such models not only streamlines the process but also ensures that smart contracts are generated with high efficiency and consistency. This advancement reduces the dependency on skilled developers, making decentralized applications more accessible to a wider range of users and organizations. Furthermore, the demonstrated feasibility of Code Llama – Instruct in smart contract auto-generation paves the way for further enhancements in contract security, optimization, and adaptability for various DAA use cases beyond those tested in this study. Future research can refine this approach by improving model training with more diverse datasets and incorporating security-focused pretraining techniques.

The functionality of the AI-generated smart contracts has been rigorously tested and validated through unit testing across four different scenarios: cryptocurrency token, ownership, wallet whitelister, and company information. In all cases, the smart contracts produced by the fine-tuned model not only passed functional validation but also exhibited lower gas fees than their human-developed counterparts. This improvement in gas efficiency translates to reduced transaction costs and improved scalability of decentralized applications, making them more sustainable in real-world deployments. Additionally, the time required for generating these contracts is significantly shorter than traditional human development, with the longest generation time recorded at just 47 seconds for the cryptocurrency token contract. This rapid generation capability enhances the agility of smart contract deployment, enabling businesses and developers to iterate and deploy contracts more swiftly. Moreover, the lower bytecode size of AI-generated contracts indicates optimized code execution, potentially contributing to reduced blockchain storage requirements and improved overall network efficiency. These advantages highlight the potential applicability of AI-generated smart contracts in a broader range of decentralized financial services, supply chain automation, and identity verification systems.

The security analysis of AI-generated smart contracts using Slither identified slightly more vulnerabilities compared to human-developed contracts, which remains a challenge in fully automating the smart contract generation process. The primary limitation stems from the smart contract dataset used in the model's training and fine-tuning process, which lacks comprehensive security-focused features. Additionally, the use of low-temperature parameters in code generation led to highly deterministic outputs, increasing predictability and potentially limiting the diversity of generated code structures. However, these limitations are not insurmountable. Increasing the temperature parameter, refining dataset selection, and incorporating adversarial training techniques could improve the robustness of auto-generated contracts against security threats. In the long term, integrating AI-driven security auditing tools directly into the generation pipeline could further enhance contract resilience. Beyond the four tested scenarios, this approach could be extended to other domains requiring automated contract execution, such as real estate transactions, insurance policies, and governance mechanisms in decentralized organizations. By continuously refining AI-driven smart contract generation, future implementations can achieve a higher level of security and adaptability, further reducing the need for human intervention in the development and maintenance of decentralized applications.

Future research should integrate advanced security verification methods during the fine-tuning process. This may involve utilizing both static and dynamic analysis techniques to detect vulnerabilities

proactively and refining the model to embed secure coding standards. Expanding the training dataset to include a broader spectrum of security threats and real-world attack scenarios would enhance the model's ability to generate more resilient smart contracts. Additionally, incorporating reinforcement learning with security-driven reward mechanisms can guide the model toward producing safer and more efficient contract code. By implementing these improvements, AI-generated smart contracts can achieve greater robustness against security risks while maintaining their efficiency and automation advantages.

6 Conclusion

This paper establishes a groundbreaking framework for AI-powered smart contract generation tailored to DAAs, demonstrating that large language models can effectively automate and optimize smart contract development. The fine-tuning of the 7-billion-parameter Code Llama – Instruct model with a meticulously curated dataset of 6,003 human instruction-Solidity pairs showcases its capability to generate fully functional smart contracts in seconds, significantly reducing both gas costs and development time. These findings reaffirm the feasibility of integrating AI into blockchain ecosystems, strengthening the core principles of decentralization and automation while setting the foundation for more sophisticated AI-assisted contract development in the future.

The key contributions of this research include the development of a high-quality dataset for smart contract generation, the fine-tuned Code Llama – Instruct model specialized in Solidity, and an extensive evaluation across four distinct smart contract scenarios. Despite its advantages, the study acknowledges security as a key challenge, as AI-generated contracts still exhibit vulnerabilities comparable to human-developed ones. Addressing this limitation requires enhancing the training dataset with more security-focused examples and leveraging more computational power to fine-tune larger models for improved performance and optimization. By overcoming these challenges, AI-driven smart contract generation can revolutionize the blockchain landscape, providing a scalable, efficient, and increasingly secure alternative to traditional contract development.

Beyond security enhancements, future work should explore the scalability of AI-generated smart contracts for larger, more complex applications. As decentralized ecosystems grow, the ability to efficiently generate and deploy intricate contracts involving multi-party agreements, automated governance, and decentralized finance (DeFi) mechanisms will be critical. Collaborations with blockchain developers, financial institutions, and governance organizations can provide valuable insights into refining AI models for real-world applications. Additionally, integrating AI-driven contract generation with existing smart contract development frameworks could create hybrid approaches that leverage both human expertise and machine efficiency, optimizing both security and functionality.

Acknowledgements

The authors would like to thank Universitas Multimedia Nusantara for the support of this study and the reviewers for their valuable insights and constructive feedback.

Data Availability Statements

The created smart contract instruction dataset can be accessed from the HuggingFace platform (https://huggingface.co/datasets/AlfredPros/smart-contracts-instructions).

The finetuned Code LLaMA - Instruct model can be accessed through the HuggingFace platform (https://huggingface.co/AlfredPros/CodeLlama-7b-Instruct-Solidity).

References

[1] Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., ... & McGrew, B. (2023). Gpt-4 technical report.

[2] Ainslie, J., Lee-Thorp, J., De Jong, M., Zemlyanskiy, Y., Lebrón, F., & Sanghai, S. (2023). Gqa: Training generalized multi-query transformer models from multi-head checkpoints.

[3] Allouche, M., Mitrea, M., Moreaux, A., & Kim, S. K. (2021). Automatic smart contract generation for internet of media things. *ICT Express*, *7*(3), 274-277. https://doi.org/10.1016/j.icte.2021.08.009.

[4] Almakhour, M., Sliman, L., Samhat, A. E., & Mellouk, A. (2020). Verification of smart contracts: A survey. *Pervasive and Mobile Computing*, *67*, 101227. https://doi.org/10.1016/j.pmcj.2020.101227.

[5] Anil, R., Dai, A. M., Firat, O., Johnson, M., Lepikhin, D., Passos, A., ... & Wu, Y. (2023). Palm 2 technical report.

[6] Augustin, N., Eckhardt, A., & de Jong, A. W. (2023). Understanding decentralized autonomous organizations from the inside. *Electronic Markets*, *33*(1), 38. https://doi.org/10.1007/s12525-023-00659-y.

[7] Chen, J., Xia, X., Lo, D., Grundy, J., Luo, X., & Chen, T. (2020). Defining smart contract defects on ethereum. *IEEE Transactions on Software Engineering*, *48*(1), 327-345. https://doi.org/10.1109/TSE.2020.2989002.

[8] Chen, Y., & Bellavitis, C. (2020). Blockchain disruption and decentralized finance: The rise of decentralized business models. *Journal of Business Venturing Insights*, *13*, e00151. https://doi.org/10.1016/j.jbvi.2019.e00151.

[9] Chu, H., Zhang, P., Dong, H., Xiao, Y., Ji, S., & Li, W. (2023). A survey on smart contract vulnerabilities: Data sources, detection and repair. *Information and Software Technology*, *159*, 107221. https://doi.org/10.1016/j.infsof.2023.107221.

[10] Chung, H. W., Hou, L., Longpre, S., Zoph, B., Tay, Y., Fedus, W., ... & Wei, J. (2024). Scaling instruction-finetuned language models. *Journal of Machine Learning Research*, *25*(70), 1-53.

[11] Dabre, R., Song, H., Exel, M., Buschbeck, B., Eschbach-Dymanus, J., & Tanaka, H. (2024, September). How Effective is Synthetic Data and Instruction Fine-tuning for Translation with Markup using LLMs?. In *Proceedings of the 16th Conference of the Association for Machine Translation in the Americas (Volume 1: Research Track)* (pp. 73-87).

[12] Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems*, *36*, 10088-10115. https://doi.org/10.48550/arXiv.2305.09808.

[13] Dingman, W., Cohen, A., Ferrara, N., Lynch, A., Jasinski, P., Black, P. E., & Deng, L. (2019). Defects and vulnerabilities in smart contracts, a classification using the NIST bugs framework. *International Journal of Networked and Distributed Computing*, *7*(3), 121-132. https://doi.org/10.2991/ijndc.k.190710.003.

[14] Feist, J., Grieco, G., & Groce, A. (2019, May). Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)* (pp. 8-15). IEEE. https://doi.org/10.1109/WETSEB.2019.00008

[15] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages.

[16] Ghaleb, A., & Pattabiraman, K. (2020, July). How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis* (pp. 415-427).

[17] Gholami, S., & Omar, M. (2023). Does synthetic data make large language models more efficient?.

[18] Hassooni, M. N. (2024). The Impact of Quantum Computing on Artificial Intelligence: An Overview. *International Academic Journal of Science and Engineering, 11*(1), 221–228. https://doi.org/10.9756/IAJSE/V11I1/IAJSE1125.

[19] Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., ... & Chen, W. (2022). Lora: Low-rank adaptation of large language models. *ICLR*, *1*(2), 3. https://arxiv.org/pdf/2106.09685v1/1000.

[20] Jain, S. M. (2022). Introduction to remix IDE. In *A Brief Introduction to Web3: Decentralized Web Fundamentals for App Development* (pp. 89-126). Berkeley, CA: Apress. https://doi.org/10.1007/978-1-4842-8975-4_5.

[21] Jain, S. M. (2022). *A Brief Introduction to Web3: Decentralized Web Fundamentals for App Development*. Apress. https://doi.org/10.1007/978-1-4842-8975-4.

[22] Jiao, J., Kan, S., Lin, S. W., Sanan, D., Liu, Y., & Sun, J. (2020, May). Semantic understanding of smart contracts: Executable operational semantics of solidity. In *2020 IEEE Symposium on Security and Privacy (SP)* (pp. 1695-1712). IEEE. https://doi.org/10.1109/SP40000.2020.00066.

[23] Kamaraj, A., & Amudha, A. (2017). PFC-Speed Controlling of BLDC Motor Using Modified BL-LUO Converter for Industrial Applications. *International Journal of Advances in Engineering and Emerging Technology*, *8*(4), 127-139.

[24] Karanjai, R., Li, E., Xu, L., & Shi, W. (2023, October). Who is smarter? an empirical study of ai-based smart contract creation. In *2023 5th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)* (pp. 1-8). IEEE. https://doi.org/10.1109/BRAINS59668.2023.10316829.

[25] Liu, C. G., Bodorik, P., & Jutla, D. (2022). Automating smart contract generation on blockchains using multi-modal modeling. *Journal of Advances in Information Technology*, *13*(3). https://doi.org/10.12720/jait.13.3.213-223.

[26] Liu, R., Wei, J., Liu, F., Si, C., Zhang, Y., Rao, J., ... & Dai, A. M. (2024). Best practices and lessons learned on synthetic data.

[27] llama-2-7b-int4-python-coder. (2023). https://huggingface.co/edumunozsala.

[28] Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., ... & Jiang, D. (2023). Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

[29] Mangrulkar, S., Gugger, S., Debut, L., Belkada, Y., Paul, S., & Bossan, B. (2022). Peft: State-of-the-art parameter-efficient fine-tuning methods. In *Peft: State-of-the-art parameter-efficient fine-tuning methods*.

[30] Marchesi, L., Mannaro, K., Marchesi, M., & Tonelli, R. (2022). Automatic generation of ethereum-based smart contracts for agri-food traceability system. *Ieee Access*, *10*, 50363-50383. https://doi.org/10.1109/ACCESS.2022.3171045.

[31] Meinhardt, D., & Krein, K. (2024). Novel approaches in AI processing systems for their better reliability and function. *International Journal of Communication and Computer Technologies*, *12*(2), 21-30. https://doi.org/10.31838/IJCCTS/12.02.03

[32] Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., ... & Dinaburg, A. (2019, November). Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 1186-1189). IEEE. https://doi.org/10.1109/ASE.2019.00133.

[33] Nelaturu, K., Beillahi, S. M., Long, F., & Veneris, A. (2021, September). Smart contracts refinement for gas optimization. In *2021 3rd conference on blockchain research & applications for innovative networks and services (BRAINS)* (pp. 229-236). IEEE. https://doi.org/10.1109/BRAINS52497.2021.9569819.

[34] Peng, B., Li, C., He, P., Galley, M., & Gao, J. (2023). Instruction tuning with gpt-4.

[35] Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., ... & Synnaeve, G. (2023). Code llama: Open foundation models for code.

[36] Sadulla, S. (2024). Next-Generation Semiconductor Devices: Breakthroughs in Materials and Applications. *Progress in Electronics and Communication Engineering*, *1*(1), 13-18. https://doi.org/10.31838/PECE/01.01.03

[37] Santana, C., & Albareda, L. (2022). Blockchain and the emergence of Decentralized Autonomous Organizations (DAOs): An integrative model and research agenda. *Technological Forecasting and Social Change*, *182*, 121806. https://doi.org/10.1016/j.techfore.2022.121806.

[38] Sayeed, S., Marco-Gisbert, H., & Caira, T. (2020). Smart contract: Attacks and protections. *Ieee Access*, *8*, 24416-24427. https://doi.org/10.1109/ACCESS.2020.2970495.

[39] Slither Audited Smart Contracts Dataset. (2022). https://huggingface.co/datasets/mwritescode/slither-audited-smart-contracts.

[40] Slither, the smart contract static analyzer. (2023). https://crytic.github.io/slither/slither.html.

[41] Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., & Alexandrov, Y. (2018, May). Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain* (pp. 9-16). https://doi.org/10.1145/3194113.3194115.

[42] Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., ... & Scialom, T. (2023). Llama 2: Open foundation and fine-tuned chat models.

[43] Wang, Y., Wang, W., Joty, S., & Hoi, S. C. (2021). Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation.

[44] Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, *151*(2014), 1-32.

[45] Xu, C., Sun, Q., Zheng, K., Geng, X., Zhao, P., Feng, J., ... & Jiang, D. (2023). Wizardlm: Empowering large language models to follow complex instructions.

[46] Ye, J., Ma, M., Peng, T., Peng, Y., & Xue, Y. (2019, April). Towards automated generation of bug benchmark for smart contracts. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (pp. 184-187). IEEE. https://doi.org/10.1109/ICSTW.2019.00049.

[47] Yuan, Z., Liu, J., Zi, Q., Liu, M., Peng, X., & Lou, Y. (2023). Evaluating instruction-tuned large language models on code comprehension and generation.

[48] Zain, Z. (2025). Exploring the field of mechatronics: Scope and future. *Innovative Reviews in Engineering and Science*, *2*(1), 45-51. https://doi.org/10.31838/INES/02.01.05

[49] Zetzsche, D. A., Arner, D. W., & Buckley, R. P. (2020). Decentralized finance. *Journal of Financial Regulation*, *6*(2), 172-203. https://doi.org/10.1093/jfr/fjaa010.

[50] Zhang, S., & Lee, J. H. (2019). Smart contract-based secure model for miner registration and block validation. *IEEE Access*, *7*, 132087-132094. https://doi.org/10.1109/ACCESS.2019.2940551.

[51] Zhao, X., Ai, P., Lai, F., Luo, X., & Benitez, J. (2022). Task management in decentralized autonomous organization. *Journal of Operations Management*, *68*(6-7), 649-674.

[52] Zheng, Z., Xie, S., Dai, H. N., Chen, W., Chen, X., Weng, J., & Imran, M. (2020). An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems*, *105*, 475-491. https://doi.org/10.1016/j.future.2019.12.019

Authors Biography

Alfred Kuhlman has earned a Bachelor's degree from Universitas Multimedia Nusantara, Indonesia, in 2024. He has been invited as a reviewer and reviewed several papers for IEEE in 2024. Currently, Alfred is working as a software developer at Indomobil Sukses Internasional. His interesting in a wide range of computer science has led him to research and develop variety of computer topics, such as blockchain, artificial intelligence, algorithm design, web and game development.

Arya Wicaksana is an Associate Professor and Head of the Department of Informatics at Universitas Multimedia Nusantara (UMN), Indonesia. He holds a Master's degree in VLSI Engineering from Universiti Tunku Abdul Rahman (UTAR), Malaysia, where he contributed to the institution's first successful ASIC design methodology for a multiprocessor system-on-chip project. His research focuses on blockchain applications and computational intelligence, particularly decentralized autonomous systems. With extensive industrial experience as an IC Design Engineer in Malaysia and China, he has worked on complex semiconductor projects and silicon tape-outs. Arya actively engages in professional communities such as IEEE and ACM, serving as a reviewer and author for scientific publications. He is also a certified Oracle Database Administrator (OCA) 11g and an EC-Council Certified Ethical Hacker (CEH), reflecting his strong technological expertise.